Portal Gate - Project

Smart Contract Security Assessment

Prepared by: **Halborn**

Date of Engagement: **August 21st, 2023 - September 1st, 2023**

Visit: **Halborn.com**

DOCUMENT REVISION HISTORY

VERSION	MODIFICATION	DATE
0.1	Document Creation	08/25/2023
0.2	Document Updates	08/25/2023
0.3	Document Updates	08/25/2023
0.4	Draft Review	08/25/2023
0.5	Draft Review	08/31/2023
1.0	Remediation Plan	09/06/2023
1.1	Remediation Plan Review	09/06/2023

CONTACTS

CONTACT	COMPANY	EMAIL
Rob Behnke	Halborn	Rob.Behnke@halborn.com
Steven Walbroehl	Halborn	Steven.Walbroehl@halborn.com
Gabi Urrutia	Halborn	Gabi.Urrutia@halborn.com
Gokberk Gulgun	Halborn	Gokberk.Gulgun@halborn.com

EXECUTIVE OVERVIEW

1.1 INTRODUCTION

Portal Gate engaged Halborn to conduct a security assessment on their smart contracts beginning on August 21st, 2023 and ending on September 1st, 2023. The security assessment was scoped to the smart contracts provided to the Halborn team.

1.2 ASSESSMENT SUMMARY

The team at Halborn was provided two weeks for the engagement and assigned a full-time security engineer to verify 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 assessment is to:

- Ensure that smart contract functions operate as intended.
- Identify potential security issues with the smart contracts.

In summary, Halborn identified some security risks that were mostly addressed by the Portal Gate team.

1.3 SCOPE

1. IN-SCOPE:

The security assessment was scoped to the following smart contracts:

- contracts/portalgate/InstanceRegistry.sol
- contracts/portalgate/KycERC20.sol
- contracts/portalgate/KycETH.sol
- contracts/portalgate/PortalGateAnonymityPoints.sol
- contracts/portalgate/PGRouter.sol
- contracts/portalgate/RelayerAggregator.sol
- contracts/portalgate/RelayerRegistry.sol
- contracts/portalgate/Zapper.sol
- /contracts/tornado-core/ERC20Tornado.sol
- /contracts/tornado-core/ETHTornado.sol
- /contracts/tornado-core/Echoer.sol
- /contracts/tornado-core/MerkleTreeWithHistory.sol
- /contracts/tornado-core/MerkleTreeWithHistoryPoseidon.sol
- /contracts/tornado-core/Miner.sol
- /contracts/tornado-core/RewardSwap.sol
- /contracts/tornado-core/Tornado.sol
- /contracts/tornado-core/TornadoTrees.sol
- /contracts/tornado-core/Verifier.sol

Commit ID: ccf60183d5027658de8d687a24a4a152ee41428b

REMEDIATION COMMIT ID :

- bbd8db93c6e08e89c0bbd1ffe6a9f8a103004e3c

1.4 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 assessment. 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 assessment:

- 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. (Hardhat)

2. RISK METHODOLOGY

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

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

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

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

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

2.1 EXPLOITABILITY

Attack Origin (AO):

Captures whether the attack requires compromising a specific account.

Attack Cost (AC):

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

Attack Complexity (AX):

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

Metrics:

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

Exploitability E is calculated using the following formula:

$$E = \prod m_e$$

2.2 IMPACT

Confidentiality (C):

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

Integrity (I):

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

Availability (A):

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

Deposit (D):

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

Yield (Y):

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

Metrics:

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

Impact I is calculated using the following formula:

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

2.3 SEVERITY COEFFICIENT

Reversibility (R):

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

Scope (S):

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

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

Severity Coefficient C is obtained by the following product:

$$C = rs$$

The Vulnerability Severity Score S is obtained by:

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

The score is rounded up to 1 decimal places.

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

3. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
0	0	1	2	9

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
(HAL-01) INCORRECT isERC20 CHECK IN zapInEth FUNCTION CAUSES INACCURATE TOKEN VALIDATION	Medium (6.2)	SOLVED - 09/06/2023
(HAL-02) MISSING A CAP FOR PROTOCOL FEE PERCENTAGE	Low (2.5)	SOLVED - 09/06/2023
(HAL-03) USE OF UNSAFE APPROVE METHOD IN ZAPINETH FUNCTION	Low (3.1)	SOLVED - 09/06/2023
(HAL-04) 2-STEP TRANSFER OWNERSHIP MISSING	Informational (1.0)	ACKNOWLEDGED
(HAL-05) DUPLICATED KYCERC20 Contract IN THE PG CORE REPOSITORY	Informational (1.0)	SOLVED - 09/06/2023
(HAL-06) USE OF CUSTOM ERRORS MISSING	Informational (1.0)	ACKNOWLEDGED
(HAL-07) INCONSISTENCY BETWEEN FILE NAME AND CONTRACT NAME	Informational (1.0)	SOLVED - 09/06/2023
(HAL-08) MISSING EVENT WHEN UPDATING GOVERNANCE ADDRESS	Informational (1.0)	SOLVED - 09/06/2023
(HAL-09) REDUNDANT OWNABLE INHERITANCE IN KYCERC20 CONTRACT	Informational (1.0)	SOLVED - 09/06/2023
(HAL-10) FLOATING PRAGMA	Informational (0.0)	SOLVED - 09/06/2023
(HAL-11) REDUNDANT HARDHAT CONSOLE	Informational (0.0)	SOLVED - 09/06/2023
(HAL-12) CACHING LENGTH IN FOR LOOPS	Informational (0.0)	SOLVED - 09/06/2023

FINDINGS & TECH DETAILS

4.1 (HAL-01) INCORRECT isERC20 CHECK IN zapInEth FUNCTION CAUSES INACCURATE TOKEN VALIDATION - MEDIUM (6.2)

Description:

The zapInEth function contains an incorrect check on isERC20 to validate if the token is an ERC20. The issue arises because this function seems to handle operations related to Ethereum deposits, and there is a requirement set that isERC20 must be true for the function to proceed (require(isERC20, "Token is not ERC20.")). This is contradictory, as ETH itself is not an ERC20 token.

Code Location:

Listing 1

```
function zapInEth(ITornadoInstance _tornado, bytes32 _commitment,
↳ bytes calldata _encryptedNote) external payable {
    (bool isERC20, IERC20 token, , , , ) = instanceRegistry.
↳ instances(_tornado);
    require(isERC20, "Token is not ERC20.");

    address _kycEth = address(token);
    KycETH kycEth = KycETH(_kycEth);
    kycEth.depositFor{ value: msg.value }();

    uint approveAmt = kycEth.allowance(address(this), address(
↳ pgRouter));
    if (approveAmt < _tornado.denomination()) {
      kycEth.approve(address(pgRouter), _tornado.denomination());
    }

    pgRouter.deposit(_tornado, _commitment, _encryptedNote, msg.
↳ sender);
  }
```

BVSS:

AO:A/AC:L/AX:L/C:L/I:L/A:M/D:N/Y:N/R:N/S:U (6.2)

Recommendation:

If this function is solely designed for handling ETH deposits, the isERC20 check is unnecessary and should be removed. As another solution, check should be changed with the following line :

Listing 2

```
require(!isERC20, "Token is ERC20.");
```

Remediation Plan:

SOLVED: The Portal Gate Team solved the issue by changing the require statement with the token existence check.

Commit ID: 64cd07e375657dd931081e3affb22dc812dace06

4.2 (HAL-02) MISSING A CAP FOR PROTOCOL FEE PERCENTAGE - LOW (2.5)

Description:

The function setProtocolFee allows the governance to set a new protocol fee percentage for a given instance. However, there is no upper limit or cap defined for the newFee parameter. This could potentially allow for an excessively high fee to be set, which could discourage users from interacting with the contract.

Code Location:

Listing 3: InstanceRegistry.sol

```
function setProtocolFee(ITornadoInstance instance, uint32 newFee)
 ↳ external onlyGovernance {
  instances[instance].protocolFeePercentage = newFee;
}
```

BVSS:

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

Recommendation:

Introduce a maximum limit for the protocol fee percentage to ensure that it cannot be set to an excessively high value.

Remediation Plan:

SOLVED: The Portal Gate Team solved the issue by implementing an upper bound on the fee percentage.

Commit ID: 64cd07e375657dd931081e3affb22dc812dace06

4.3 (HAL-03) USE OF UNSAFE APPROVE METHOD IN ZAPINETH FUNCTION - LOW (3.1)

Description:

The zapInEth function within the contract uses the approve method from the ERC20 standard to set the allowance for pgRouter. However, this implementation does not use the safeApprove method available in OpenZeppelin 's SafeERC20 library. The use of plain approve might lead to potential issues due to the allowance manipulation vulnerability, known as approval race condition.

Code Location:

Listing 4

```
function zapInEth(ITornadoInstance _tornado, bytes32 _commitment
    , bytes calldata _encryptedNote) external payable {
    (bool isERC20, IERC20 token, , , , ) = instanceRegistry.
    instances(_tornado);
    require(isERC20, "Token is not ERC20.");

    address _kycEth = address(token);
    KycETH kycEth = KycETH(_kycEth);
    kycEth.depositFor{ value: msg.value }();

    uint approveAmt = kycEth.allowance(address(this), address(
    pgRouter));
    if (approveAmt < _tornado.denomination()) {
      kycEth.approve(address(pgRouter), _tornado.denomination());
    }

    pgRouter.deposit(_tornado, _commitment, _encryptedNote, msg.
    sender);
  }
```

BVSS:

AO:A/AC:L/AX:L/C:L/I:L/A:N/D:N/Y:N/R:N/S:U (3.1)

Recommendation:

To mitigate the potential risks associated with the approve method, you should consider using the safeApprove method from OpenZeppelin's SafeERC20 library. This will ensure that the contract’s token operations are secure and resistant to known vulnerabilities.

Remediation Plan:

SOLVED: The Portal Gate Team solved the issue by changing approve function with safeApprove method.

Commit ID: 64cd07e375657dd931081e3affb22dc812dace06

4.4 (HAL-04) 2-STEP TRANSFER OWNERSHIP MISSING - INFORMATIONAL (1.0)

Description:

The code does not implement a two-step ownership transfer pattern. This practice is recommended when admin users have heavy responsibilities, such as the ability to mint tokens, freeze or unfreeze user accounts and set system configurations. It may happen that when transferring ownership of a contract, an error is made in the address. If the request were submitted, the contract would be lost forever. With this pattern, contract owners can submit a transfer request; however, this is not final until accepted by the new owner. If they realize they have made a mistake, they can stop it at any time before accepting it by calling cancelRequest.

Code Location:

Listing 5: RelayerRegistry.sol

```
contract RelayerRegistry is Ownable {
```

BVSS:

AO:S/AC:L/AX:L/C:N/I:N/A:C/D:N/Y:N/R:P/S:U (1.0)

Recommendation:

It is recommended to implement a two-step process where the owner nominates an account, and the nominated account needs to call an acceptOwnership() function for the transfer of the ownership to succeed fully. This ensures the nominated EOA account is valid and active.

Remediation Plan:

ACKNOWLEDGED: The Portal Gate Team acknowledged this finding. To mitigate existing issues, The Portal Gate Team will utilize a multi-signature wallet. In forthcoming releases, administrative access will be transitioned to a Decentralized Autonomous Organization (DAO) for enhanced governance.

4.5 (HAL-05) DUPLICATED KYCERC20 Contract IN THE PG CORE REPOSITORY - INFORMATIONAL (1.0)

Description:

The KycERC20.sol contract appears to be duplicated within the pg-core-contracts-v1 repository. Specifically, the contract is present in both the pg-core-contracts-v1/contracts/keyring/tokens/ directory and the pg-core-contracts-v1/contracts/portalgate/ directory. This could lead to confusion, increase the chance of bugs, and make the codebase harder to maintain.

Code Location:

Listing 6

```

pragma solidity ^0.8.0;

import "../keyring/tokens/KycERC20.sol";
import "./KycETH.sol";
import "../interfaces/ITornadoInstance.sol";
import "./PGRouter.sol";
import "./InstanceRegistry.sol";

import "@openzeppelin/contracts/token/ERC20/utils/SafeERC20.sol";

import "@openzeppelin/contracts/token/ERC20/ERC20.sol";
import "@openzeppelin/contracts/token/ERC20/IERC20.sol";

contract Zapper {}
```

BVSS:

AO:S/AC:L/AX:L/C:N/I:N/A:C/D:N/Y:N/R:P/S:U (1.0)

Recommendation:

If the contracts are identical, consider keeping only one copy in a common directory that both keyring/tokens and portalgate can reference, thereby removing the duplicate code.

Remediation Plan:

SOLVED: The Portal Gate Team solved the issue by deleting the redundant contract.

Commit ID: 64cd07e375657dd931081e3affb22dc812dace06

4.6 (HAL-06) USE OF CUSTOM ERRORS MISSING - INFORMATIONAL (1.0)

Description:

Failed operations in this contract are reverted with an accompanying message selected from a set of hard-coded strings.

In EVM, emitting a hard-coded string in an error message costs ~50 more gas than emitting a custom error. Additionally, hard-coded strings increase the gas required to deploy the contract.

Code Location:

Listing 7: PGRouter.sol (Lines 74,75)

```
65 function deposit(
66     ITornadoInstance _tornado,
67     bytes32 _commitment,
68     bytes memory _encryptedNote,
69     address sender
70   ) public payable virtual {
71     (bool isERC20, IERC20 token, InstanceRegistry.InstanceState
 ↳ state, , , uint256 maxDepositAmount) = instanceRegistry.instances(
72       _tornado
73     );
74     require(state != InstanceRegistry.InstanceState.DISABLED, "The
 ↳  instance is not supported");
75     require(token.balanceOf(address(_tornado)) < maxDepositAmount,
 ↳  "Exceed deposit Cap for the pool");
76
77     if (isERC20) {
78       token.safeTransferFrom(msg.sender, address(this), _tornado.
 ↳ denomination());
79     }
80     _tornado.deposit{ value: msg.value }(_commitment);
81
82     if (state == InstanceRegistry.InstanceState.MINABLE) {
83       tornadoTrees.registerDeposit(address(_tornado), _commitment)
 ↳ ;
```

```
    }

    emit EncryptedNote(sender, _encryptedNote);
}
```

BVSS:

AO:S/AC:L/AX:L/C:N/I:N/A:C/D:N/Y:N/R:P/S:U (1.0)

Recommendation:

Custom errors are available from Solidity version 0.8.4 up. Consider replacing all revert strings with custom errors. Usage of custom errors should look like this:

Listing 8

```
error CustomError();

// ...

if (condition)
    revert CustomError();
```

Remediation Plan:

ACKNOWLEDGED: The Portal Gate Team acknowledged this finding.

4.7 (HAL-07) INCONSISTENCY BETWEEN FILE NAME AND CONTRACT NAME - INFORMATIONAL (1.0)

Description:

The file name and the contract name within the Solidity file are inconsistent. This can lead to confusion and potential errors when trying to interact with or deploy the contract. In Solidity, it is a best practice to keep the contract name and the file name the same for clarity and ease of management.

Code Location:

Listing 9: PGAP.sol

```
contract PortalGateAnonymityPoints is ERC20, ERC20Burnable,
↳ Pausable, Ownable {
```

BVSS:

AO:S/AC:L/AX:L/C:N/I:N/A:C/D:N/Y:N/R:P/S:U (1.0)

Recommendation:

Rename either the file or the contract so that they match. If the contract name is PortalGateAnonymityPoints, then the file name should ideally be PortalGateAnonymityPoints.sol.

Remediation Plan:

SOLVED: The Portal Gate Team solved the issue by changing the contract name.

Commit ID: 64cd07e375657dd931081e3affb22dc812dace06

4.8 (HAL-08) MISSING EVENT WHEN UPDATING GOVERNANCE ADDRESS - INFORMATIONAL (1.0)

Description:

The setNewGovernance function in the InstanceRegistry.sol, PGRouter.sol, and Zapper.sol contracts updates the governance address but does not emit an event to log this significant change.

Code Location:

Listing 10: InstanceRegistry.sol

```
function setNewGovernance(address _govAddr) external
 ↳ onlyGovernance {
     governance = _govAddr;
   }
```

Listing 11: PGRouter.sol

```
function setNewGovernance(address _govAddr) external
 ↳ onlyGovernance {
   governance = _govAddr;
}
```

Listing 12: Zapper.sol

```
function setNewGovernance(address _govAddr) external
 ↳ onlyGovernance  {
   governance = _govAddr;
}
```

BVSS:

AO:S/AC:L/AX:L/C:N/I:N/A:C/D:N/Y:N/R:P/S:U (1.0)

Recommendation:

Consider omitting an event whenever the governance address is updated. Define an event like GovernanceUpdated and emit it in the setNewGovernance function after successfully updating the governance address.

Remediation Plan:

SOLVED: The Portal Gate Team solved the issue by emitting events on the functions.

Commit ID: 64cd07e375657dd931081e3affb22dc812dace06

4.9 (HAL-09) REDUNDANT OWNABLE INHERITANCE IN KYCERC20 CONTRACT - INFORMATIONAL (1.0)

Description:

The KycERC20 contract imports the Ownable contract from OpenZeppelin but does not actually inherit from it.

Code Location:

Listing 13: InstanceRegistry.sol

```
import "../keyring/interfaces/IKycERC20.sol";
import "../keyring/integration/KeyringGuard.sol";
import "@openzeppelin/contracts/access/Ownable.sol";
import "@openzeppelin/contracts/token/ERC20/extensions/ERC20Permit
  ↳ .sol";
import "@openzeppelin/contracts/token/ERC20/extensions/
  ↳ ERC20Wrapper.sol";
import "@openzeppelin/contracts/token/ERC20/utils/SafeERC20.sol";


contract KycERC20 is IKycERC20, ERC20Permit, ERC20Wrapper,
  ↳ KeyringGuard {}
```

BVSS:

AO:S/AC:L/AX:L/C:N/I:N/A:C/D:N/Y:N/R:P/S:U (1.0)

Recommendation:

Consider removing the import statement for Ownable to clean up the code.

Remediation Plan:

SOLVED: The Portal Gate Team solved the issue by deleting redundant inheritance.

Commit ID: 64cd07e375657dd931081e3affb22dc812dace06

4.10 (HAL-10) FLOATING PRAGMA - INFORMATIONAL (0.0)

Description:

Contracts should be deployed with the same compiler version and flags used during development and testing. Locking the pragma helps to ensure that contracts do not accidentally get deployed using another pragma. For example, an outdated pragma version might introduce bugs that affect the contract system negatively.

Code Location:

All the contracts in the repository are using the `pragma solidity ^0.8.19;` floating pragma.

BVSS:

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

Recommendation:

Consider locking the pragma version in the smart contracts. It is not recommended to use a floating pragma in production.

For example: `pragma solidity 0.8.19;`

Remediation Plan:

SOLVED: The Portal Gate Team solved the issue by locking pragma to 0.8.14.

Commit ID: 64cd07e375657dd931081e3affb22dc812dace06

4.11 (HAL-11) REDUNDANT HARDHAT CONSOLE - INFORMATIONAL (0.0)

Description:

The smart contract imports the Hardhat console library using import hardhat/console.sol;. While the Hardhat console is useful for debugging during the development and testing phases, it is generally not recommended to be included in the production code. The inclusion of debugging tools in production can lead to additional gas costs.

Code Location:

TornadoTrees.sol#L9

BVSS:

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

Recommendation:

For a cleaner, more secure, and efficient production environment, it is recommended to remove the Hardhat console import in the final version of the contract.

Remediation Plan:

SOLVED: The Portal Gate Team solved the issue by deleting console import.

Commit ID: 64cd07e375657dd931081e3affb22dc812dace06

4.12 (HAL-12) CACHING LENGTH IN FOR LOOPS - INFORMATIONAL (0.0)

Description:

In a for loop, the length of an array can be put in a temporary variable to save some gas. This has been done already in several other locations in the code.

In the above case, the solidity compiler will always read the length of the array during each iteration. That is,

- if it is a storage array, this is an extra sload operation (100 additional extra gas (EIP-2929) for each iteration except for the first),
- if it is a memory array, this is an extra mload operation (3 additional gas for each iteration except for the first),
- if it is a calldata array, this is an extra calldataload operation (3 additional gas for each iteration except for the first)

Code Location:

Listing 14

```
function relayersData(address[] memory _relayers) public view
↳ returns (Relayer[] memory) {
  Relayer[] memory relayers = new Relayer[](_relayers.length);

  for (uint256 i = 0; i < _relayers.length; i++) {
    relayers[i].isRegistered = relayerRegistry.
↳ isRelayerRegistered(_relayers[i]);
    relayers[i].balance = 0;
  }

  return relayers;
}
```

BVSS:

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

Recommendation:

In a for loop, store the length of an array in a temporary variable.

Remediation Plan:

SOLVED: The Portal Gate Team solved the issue by caching array length.

Commit ID: 64cd07e375657dd931081e3affb22dc812dace06

AUTOMATED TESTING

5.1 AUTOMATED SECURITY SCAN

Description:

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

MythX results:

- No major issues found by Mythx.

THANK YOU FOR CHOOSING

HALBORN