CHAIN TROOPERS

Hilter
Cross-Chain Gateway Protocol (CGP)
Security Assessment Report

August 3 , 2025

Version 1.1

CONFIDENTIAL

Table of Contents

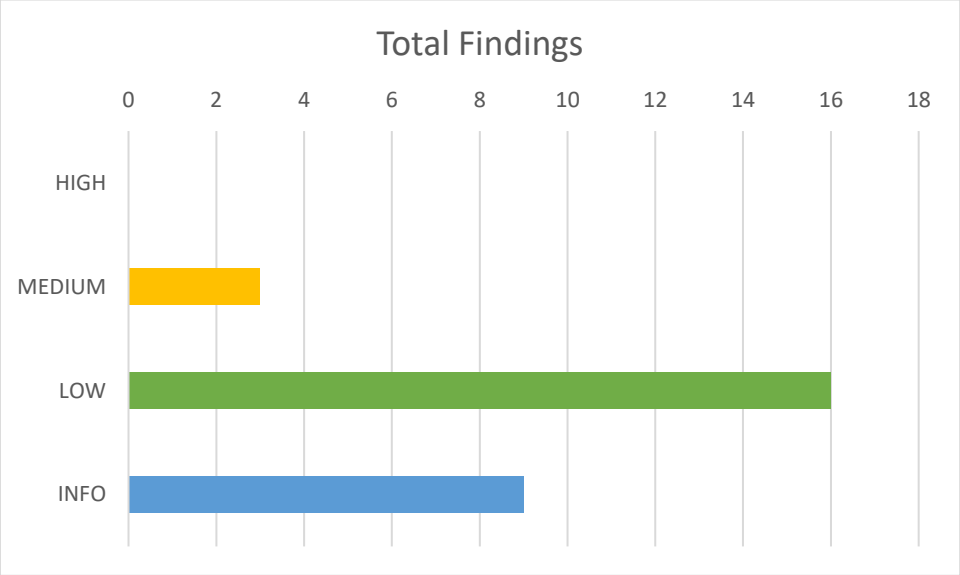
1 Executive Summary

1.1 Introduction

The report contains the results of Hilter Cross-Chain Gateway Protocol security assessment that took place from June 20th, 2025, to July 12th, 2025 and from July 15th, 2025 to July 17, 2025.th The security engineers performed an in-depth manual analysis of the provided functionalities, and uncovered issues that may be used by adversaries to affect the confidentiality, the integrity, and the availability of the in-scope components.

All the identified vulnerabilities are presented in the report, including their impact and the proposed mitigation strategy, and are ordered by their severity.

In total, the team identified nineteen (19) vulnerabilities. There were also nine (9) informational issues of no-risk.



All the identified vulnerabilities are presented in the report, including their impact and the proposed mitigation strategy, and are ordered by their severity. A retesting phase was carried out on August 2nd, 2025, and the results are presented in Section 6.

1.2 Assessment Results

The assessment results revealed that the in-scope application components were mainly vulnerable to three (3) Data Validation issues of MEDIUM risk. More precisely, it was identified that the"_transferOwnership " functionality does not impose an upper bound for the assigned weights on the selected operators *('5.1.1 - No upper bound in one operator 's weight at "HilterAuthWeighted .sol'"*), allowing values that can be excessive or even more than the assigned threshold. The team also identified that the "_transferOwnership" functionality does not enforce a lower bound for the selected new threshold (*'5.1.2 - No lower bound in threshold at "Hilter AuthWeighted .sol"'*), permitting values that can be lower than the maximum weight that has been selected for one of the operators.

Furthermore, it was found that the function which is used by administrators to set the token's limits, contains a potentially costly loop that makes the function inefficient for using it in emergency cases *('5.1.3 - Excessive loop iterations allowed in "setTokenDailyMintLimits " at "Hilter Gateway .sol"').* If the admins provide a significantly large array of tokens symbols, it is possible that the function will not be fully executed neither in the current nor in the following blocks.

There were also fifteen (15) vulnerabilities of LOW risk and seven (7) findings of no-risk (INFORMATIONAL). Regarding the Administration issues of LOW risk, it was found that many admin functionalities do not emit the appropriate event when the native token is selected *('5.2.3 - Event not emitted in self functionality " collectFees()" at "HilterGasService.sol"', '5.2.4 - Event not emitted in self functionality " refund ()" at "Hilter GasService .sol"'),* potentially affecting the credibility and the confidence in the system. A similar issue occurs when an ERC20 token is used *('5.2.1 - Event not emitted in "burnToken " functionality at "Hilter Gateway .sol"', '5.2.2 - Event not emitted in "mintToken" functionality at "HilterGateway.sol"'),* even though fully compliant ERC20 tokens should typically emit a related event. In reference to the Access Control LOW-risk issues, it was found that the contracts do not have a dedicated circuit breaker control that can be used in case of emergency to pause the transactions *('5.2.6 - Lack of circuit breaker for emergency stop at "HilterDepositService"', '5.2.7 - Lack of circuit breaker for emergency stop at*

"HilterGateway"). There is only one control based on the token limits that might not be effective as described in *finding 5.1.3.*

In reference to the LOW-risk Authentication issues, it was found that the admin functions of one contract that provides an upgrade mechanism is not protected with multisig *('5.2.11 - No multisig protection in "util/upgradable .sol"),* allowing adversaries who have access to the admin's private key to fully compromise the related contracts. Regarding the Data Validation issues of LOW risk, it was found that many external functionalities do not validate if the address of the receiver is zero *('5.2.14 - Unvalidated address "receiver" in "collectFees" at "HilterGasService.sol"' '5.2.15 - Unvalidated address "receiver" in "refund" at "HilterGasService.sol"', '5.2.16 - Unvalidated address "recipient" in "withdrawNative" at "HilterDepositService.sol")* or if the provided amount is zero *('5.2.8 - Unvalidated amount in "_mintToken " at " Hilter Gateway .sol"', '5.2.9 - Unvalidated amount in "burn" and "burnFrom " at " BurnableMintableCappedERC 20.sol"' '5.2.10 - Unvalidated amount in "refund" at " Hilter GasService .sol"', '5.2.12 - Unvalidated amount in "payGasForContractCall ()", " payGasForContractCallWithToken ()", and the "addGas ()" functions at "Hilter GasService .sol"', '5.2.13 - Unvalidated amount in "collectfees" at "HilterGasService. sol"),* facilitating user mistakes that could accidentally burn tokens , or consume unnecessary gas, while emitting confusing events for front- end dapps. Moreover, it was found that many functionalities of the Deposit Service replace the receiving address with the "msg.sender" when the provided address is zero ('*5.2.5 - Insecure error handling of zero addresses at "ReceiverImplementation .sol" and at "DepositReceiver.sol"*').

1.2.1 Retesting Results

Results from retesting carried out on August 2025, determined that four (4) reported LOW-risk issues (see sections *5.2.13, 5.2.14, 5.2.15, 5.2.16*) and one (1) INFORMATIONAL issue (see sections *5.3.7*) were sufficiently addressed (5 out of 28 findings).

1.3 Summary of Findings

The following findings were identified in the examined source code:

Vulnerability Name	Status	Retest Status	Page
No upper bound in one operator's weight at "HilterAuthWeighted.sol"	N/A	MEDIUM	15
No lower bound in threshold at "HilterAuthWeighted.sol"	N/A	MEDIUM	17
Excessive loop iterations allowed in "setTokenDailyMintLimits" at “HilterGateway.sol"	MEDIUM	MEDIUM	19
Event not emitted in "burnToken" functionality at "HilterGateway.sol"	LOW	LOW	22
Event not emitted in "mintToken" functionality at "HilterGateway.sol"	LOW	LOW	27
Event not emitted in self functionality "collectFees()" at "HilterGasService.sol"	LOW	LOW	33
Event not emitted in self functionality "refund()" at "HilterGasService.sol"	LOW	LOW	36
Insecure error handling of zero addresses at "ReceiverImplementation.sol" and at "DepositReceiver.sol"	N/A	LOW	39
Lack of circuit breaker for emergency stop at "HilterDepositService"	LOW	LOW	41
Lack of circuit breaker for emergency stop at "HilterGateway"	LOW	LOW	43
Unvalidated amount in "_mintToken" at "HilterGateway.sol"	LOW	LOW	46

Unvalidated amount in "burn" and "burnFrom" at “BurnableMintableCappedERC20.sol"	LOW	LOW	49
Unvalidated amount in "refund" at "HilterrGasService.sol"	LOW	LOW	52
No multisig protection in "util/upgradable.sol"	LOW	LOW	54
Unvalidated amount in "payGasForContractCall()", "payGasForContractCallWithToken()", and the "addGas()" functions at "HilterGasService.sol"	LOW	LOW	57
Unvalidated amount in "collectfees" at "HilterGasService.sol"	LOW	CLOSED	61
Unvalidated address "receiver" in "collectFees" at "HilterGasService.sol"	LOW	CLOSED	64
Unvalidated address "receiver" in "refund" at "HilterGasService.sol"	LOW	CLOSED	66
Unvalidated address "recipient" in "withdrawNative" at "HilterDepositService.sol"	LOW	CLOSED	68
Ownership can be transferred to same owner at "Ownable.sol"	INFO	INFO	71
Lack of circuit breaker for emergency stop at "HilterGasService"	INFO	INFO	73
Excessive loop iterations allowed in "setAdmins" at "AdminMultisigBase.sol"	INFO	INFO	75
Excessive loop iterations allowed in "admins" at "HilterGateway.sol"	INFO	INFO	78
Excessive loop iterations allowed in "collectFees" at "HilterGasService.sol"	INFO	INFO	82

Excessive loop iterations allowed in "execute" at “HilterGateway.sol"	INFO	INFO	85
No reentrancy protection in "execute" at "DepositReceiver.sol"	INFO	CLOSED	87
Floating pragma in multiple interfaces at "contracts/interfaces/" folder	N/A	INFO	89
Setup functionality can be circumvented during contract upgrade at "/contracts/util/Upgradable.sol"	N/A	INFO	91

2 Assessment Description

2.1 Target Description

Hilter network's decentralized validators confirm events emitted on EVM chains (such as deposit confirmation) and sign off on commands submitted (by automated services) to the gateway smart contracts (such as minting token, and approving message on the destination).

2.2 In-Scope Components

The components are located at the following URL:

https://gitlab.com/hilterltd-group/hilter-cgp-solidity

Component	Commit Identifier
hilter-cgp-solidity	*02dfea2e43b5d20af4c7bb0f6a2e7b045f2ad8bc*
hilter-cgp-solidity (v4.3.0) – Retest Version	*5614e209441c2f4e1b905e2746c94af2067169bc*

3 Methodology

3.1 Assessment Methodology

Chaintroopers' methodology attempts to bridge the penetration testing and source code reviewing approaches in order to maximize the effectiveness of a security assessment.

Traditional pentesting or source code review can be done individually and can yield great results, but their effectiveness cannot be compared when both techniques are used in conjunction.

In our approach, the application is stress tested in all viable scenarios though utilizing penetration testing techniques with the intention to uncover as many vulnerabilities as possible. This is further enhanced by reviewing the source code in parallel to optimize this process.

When feasible our testing methodology embraces the Test-Driven Development process where our team develops security tests for faster identification and reproducibility of security vulnerabilities. In addition, this allows for easier understanding and mitigation by development teams.

Chaintroopers' security assessments are aligned with OWASP TOP10 and NIST guidance.

This approach, by bridging penetration testing and code review while bringing the security assessment in a format closer to engineering teams has proven to be highly effective not only in the identification of security vulnerabilities but also in their mitigation and this is what makes Chaintroopers' methodology so unique.

3.2 Smart Contracts

The testing methodology used is based on the empirical study "Defining Smart Contract Defects on Ethereum" by J. Chen, X. Xia, D. Lo, J. Grundy, X. Luo and T. Chen, in IEEE Transactions on Software Engineering, and the security best practices as described in "Security Considerations" section of the solidity wiki.

The following is a non-exhaustive list of security vulnerabilities that are identified by our methodology during the examination of the in-scope contract:

- Unchecked External Calls
- Strict Balance Equality
- Transaction State Dependency
- Hard Code Address
- Nested Call
- Unspecified Compiler Version
- Unused Statement
- Missing Return Statement
- Missing Reminder
- High Gas Consumption Function Type
- DoS Under External Influence
- Unmatched Type Assignment
- Re-entrancy
- Block Info Dependency
- Deprecated APIs
- Misleading Data Location
- Unmatched ERC-20 standard
- Missing Interrupter
- Greedy Contract
- High Gas Consumption Data Type

In Substrate Pallets, the list of vulnerabilities that are identified also includes:

- Static or Erroneously Calculated Weights
- Arithmetic Overflows
- Unvalidated Inputs
- Runtime Panic Conditions
- Missing Storage Deposit Charges
- Non-Transactional Dispatch Functions
- Unhandled Errors &Unclear Return Types
- Missing Origin Authorization Checks

4 Scoring System

4.1 CVSS

All issues identified as a result of Chaintroopers' security assessments are evaluated based on Common Vulnerability Scoring System version 3.1.

With the use of CVSS, taking into account a variety of factors a final score is produced ranging from 0 up to 10. The higher the number goes the more critical an issue is.

The following table helps provide a qualitative severity rating:

Rating	CVSS Score
None/Informational	0.0
Low	0.1-3.9
Medium	4.0-6.9
High	7.0-8.9
Critical	9.0-10.0

Issues reported in this document contain a CVSS Score section, this code is provided as an aid to help verify the logic of the team behind the evaluation of a said issue.

5 Identified Findings

5.1 Medium Severity Findings

5.1.1 No upper bound in one operator's weight at "HilterAuthWeighted.sol

Description **MEDIUM**

The team identified that no upper bound is set for the provided weight for an operator at the "*_transferOwnership*" functionality. In general, the auth contract verifies that the received commands are signed by a weighted set of operator keys. It also performs transfers of operatorships (to mimic changes to the validator set of Hilter Proof-of-Stake network). However, it was found that the transfers of operatorships do not validate if the weight of an operator is excessive or even more than the required *newThreshold*.

The issue exists at:

```
File: /hilter-cgp-solidity-4.3.0/contracts/auth/HilterAuthWeighted.sol
70:         uint256 totalWeight = 0;
71:         for (uint256 i = 0; i < weightsLength; ++i) {
72:             totalWeight += newWeights[i];
73:         }
74:         ...
```

Recommendation

It is recommended to validate that the weight of each operator does not exceed an accepted fraction of the *newThreshold*.

CVSS Score

AV:N/AC:H/PR:H/UI:N/S:U/C:N/I:H/A:N/E:P/RL:X/RC:C/CR:X/IR:X/AR:X/MAV:X/MAC:X/MPR:X/MUI:X/MS:X/MC:X/MI:X/MA:X

5.1.2 No lower bound in threshold at "HilterAuthWeighted.sol"

Description **MEDIUM**

The team identified that no lower bound is set for the provided new threshold for a set of operators at the "*_transferOwnership*" functionality. In general, the auth contract verifies that the received commands are signed by a weighted set of operator keys. It also performs transfers of operatorships (to mimic changes to the validator set of Hilter Proof-of-Stake network). However, it was found that the transfers of operatorships do not validate if the required *newThreshold* is at least greater than the maximum weight that has been provided for one of the operators. For example, it is possible to provide a *newThreshold* that will be just 1, allowing any operator with weight greater than 0 to execute arbitrary commands and compromise the gateway.

The issue exists at:

```
File: /hilter-cgp-solidity-4.3.0/contracts/auth/HilterAuthWeighted.sol
70:         uint256 totalWeight = 0;
71:         for (uint256 i = 0; i < weightsLength; ++i) {
72:             totalWeight += newWeights[i];
73:         }
74:         if (newThreshold == 0 || totalWeight < newThreshold) revert
InvalidThreshold();
75:
```

Recommendation

It is recommended to validate that the *newThresold* is at least greater than the maximum weight that has been provided for one of the operators.

CVSS Score

AV:N/AC:H/PR:H/UI:N/S:U/C:N/I:H/A:N/E:P/RL:X/RC:C/CR:X/IR:X/AR:X/MAV:X/MAC:X/MPR:X/MUI:X/MS:X/MC:X/MI:X/MA:X

5.1.3 Excessive loop iterations allowed in "setTokenDailyMintLimits" at "HilterGateway.sol"

Description	MEDIUM

It was identified that the external function "setTokenDailyMintLimits", which can be called only by administrators, contains a potentially costly loop. Computational power on blockchain environments is paid, thus reducing the computational steps required to complete an operation is not only a matter of optimization but also cost efficiency. Loops are a great example of costly operations: as many elements an array has, more iterations will be required to complete the loop.

Excessive loop iterations exhaust all available gas.

In the specific case, the function "setTokenDailyMintLimits" iterates over the arrays "symbols" and "limits" which are provided as arguments and are of unspecified length:

File: hilter-cgp-solidity/contracts/HilterGateway.sol

```
204:       function setTokenDailyMintLimits(string[] calldata symbols,
uint256[] calldata limits) external override onlyAdmin {
205:                   if (symbols.length != limits.length) revert
InvalidSetDailyMintLimitsParams();
206:
207:         for (uint256 i = 0; i < symbols.length; i++) {
208:             string memory symbol = symbols[i];
209:             uint256 limit = limits[i];
210:
211:                 if (tokenAddresses(symbol) == address(0)) revert
TokenDoesNotExist(symbol);
212:
213:             _setTokenDailyMintLimit(symbol, limit);
214:         }
215:     }
```

In case that the Administrator decides to apply specific limits to a large array of symbols, it is possible that the operation will fail due to the max gas consumption on the current block. A failed change in the limits will allow adversaries who monitor the transactions to identify the requested action and use front running to circumvent the limitation before it is applied in the following blocks. Furthermore, the Administrator will have to submit the action in smaller batches to be able to execute it, allowing the adversaries to still circumvent the limits in the remaining symbols.

Recommendation

If it is necessary to loop over an array of unknown size, the function should be able to execute the operation in multiple blocks and in multiple transactions . In that case, it will be required to maintain the extra state of how many iterations have already been performed to continue from that point in the next function call. In case that there is no requirement to loop over an array of unknown size, it is advisable to modify the functionality to always verify that the provided symbols array does not exceed an upper limit to prevent a failure in the update operation.

CVSS Score

AV:N/AC:H/PR:H/UI:N/S:U/C:L/I:L/A:L/E:F/RL:X/RC:C/CR:X/IR:X/AR:X/MAV:X/MAC:X/MPR:X/MUI:X/MS:X/MC:X/MI:X/MA:X

5.2 Low Severity Findings

5.2.1 Event not emitted in "burnToken" functionality at "HilterGateway.sol"

Description	LOW

It was identified that the admin command "burnToken" does not emit an event with the exact amount when an external token is used. A contract can emit events when it wants to notify external entities like users, chain explorers, or dApps about changes or conditions in the blockchain. When an event is emitted, it stores the arguments passed in transaction logs. These logs are stored on blockchain and are accessible using address of the contract till the contract is present on the blockchain

The issue exists at the following location:

```
File: /hilter-cgp-solidity/contracts/HilterGateway.sol
373:        function burnToken(bytes calldata params, bytes32) external
onlySelf {
374:            (string memory symbol, bytes32 salt) = abi.decode(params,
(string, bytes32));
375:
376:         address tokenAddress = tokenAddresses(symbol);
377:
378:                         if (tokenAddress == address(0)) revert
TokenDoesNotExist(symbol);
379:
380:         if (_getTokenType(symbol) == TokenType.External) {
381:             DepositHandler depositHandler = new DepositHandler{ salt:
salt }();
382:
383:                         (bool success, bytes memory returnData) =
depositHandler.execute(
384:                 tokenAddress,
385:                     abi.encodeWithSelector(IERC20.transfer.selector,
address(this), IERC20(tokenAddress).balanceOf(address(depositHandler)))
386:             );
387:
```

```
388:                 if (!success || (returnData.length != uint256(0) &&
!abi.decode(returnData, (bool)))) revert BurnFailed(symbol);
389:
390:             // NOTE: `depositHandler` must always be destroyed in the
same runtime context that it is deployed.
391:             depositHandler.destroy(address(this));
392:         } else {
393:             IBurnableMintableCappedERC20(tokenAddress).burn(salt);
394:         }
395:     }
```

The admin command is parsed at the following location:

```
File: /hilter-cgp-solidity/contracts/HilterGateway.sol
262:     function execute(bytes calldata input) external override {
263:         (bytes memory data, bytes memory proof) = abi.decode(input,
(bytes, bytes));
264:
265:                                         bytes32    messageHash    =
ECDSA.toEthSignedMessageHash(keccak256(data));
266:
267:         // TEST auth and getaway separately
268:                                       bool    currentOperators    =
IHilterAuth(AUTH_MODULE).validateProof(messageHash, proof);
269:
270:         uint256 chainId;
271:         bytes32[] memory commandIds;
272:         string[] memory commands;
273:         bytes[] memory params;
274:
275:         try HilterGateway(this)._unpackLegacyCommands(data) returns (
276:             uint256 chainId_,
277:             bytes32[] memory commandIds_,
278:             string[] memory commands_,
279:             bytes[] memory params_
280:         ) {
281:                 (chainId, commandIds, commands, params) = (chainId_,
commandIds_, commands_, params_);
282:         } catch {
```

```
283:            (chainId, commandIds, commands, params) = abi.decode(data,
(uint256, bytes32[], string[], bytes[]));
284:         }
285:
286:         if (chainId != block.chainid) revert InvalidChainId();
287:
288:         uint256 commandsLength = commandIds.length;
289:
290:          if (commandsLength != commands.length || commandsLength !=
params.length) revert InvalidCommands();
291:
292:         for (uint256 i; i < commandsLength; ++i) {
293:             bytes32 commandId = commandIds[i];
294:
295:             if (isCommandExecuted(commandId)) continue; /* Ignore if
duplicate commandId received */
296:
297:             bytes4 commandSelector;
298:                                                 bytes32   commandHash   =
keccak256(abi.encodePacked(commands[i]));
299:
300:            ...
308:             } else if (commandHash == SELECTOR_BURN_TOKEN) {
 309:                 commandSelector = HilterGateway.burnToken.selector;
310:            …
```

If the internal token implementation is used, then the burn function of the "BurnableMintableCappedERC20" will be called, which indeed will emit an event as part of the open zeppelin ERC20 implementation:

```
File: /hilter-cgp-solidity/contracts/BurnableMintableCappedERC20.sol
34:     function burn(bytes32 salt) external onlyOwner {
35:         address account = depositAddress(salt);
36:         _burn(account, balanceOf[account]);
37:     }
```

```
File: /hilter-cgp-solidity/contracts/ERC20.sol
200:      * Emits a {Transfer} event with `to` set to the zero address.
201:      *
202:      * Requirements:
203:      *
204:      * - `account` cannot be the zero address.
205:      * - `account` must have at least `amount` tokens.
206:      */
207:     function _burn(address account, uint256 amount) internal virtual
{
208:         if (account == address(0)) revert InvalidAccount();
209:
210:         _beforeTokenTransfer(account, address(0), amount);
211:
212:         balanceOf[account] -= amount;
213:         totalSupply -= amount;
214:         emit Transfer(account, address(0), amount);
215:     }
```

However, in case that an external token implementation is used, it is possible that no event regarding the exact minted amount will be emitted. Currently, the DepositHandler implementation is the following and emits no event:

```
File: /hilter-cgp-solidity/contracts/DepositHandler.sol
22:       function execute(address callee, bytes calldata data) external
noReenter returns (bool success, bytes memory returnData) {
23:         if (callee.code.length == 0) revert NotContract();
24:         (success, returnData) = callee.call(data);
25:     }
```

On the other hand, an event about the successful execution of the command will be emitted by the "execute" functionality:

```
File: /hilter-cgp-solidity/contracts/HilterGateway.sol
292:         for (uint256 i; i < commandsLength; ++i) {
293:             bytes32 commandId = commandIds[i];
```

```
294:
295:             ...
317:
320:                                                (bool    success,    )    =
address(this).call(abi.encodeWithSelector(commandSelector,      params[i],
commandId));
321:
322:             if (success) emit Executed(commandId);
323:             else _setCommandExecuted(commandId, false);
```

Recommendation

It is recommended to emit an event related to this functionality.

CVSS Score

AV:N/AC:H/PR:N/UI:N/S:U/C:L/I:N/A:N/E:P/RL:X/RC:C/CR:X/IR:X/AR:X/MAV:X/MAC:X/MPR:X/MUI:X/MS:X/MC:X/MI:X/MA:X

5.2.2 Event not emitted in "mintToken" functionality at "HilterGateway.sol"

Description	LOW

It was identified that the admin command "mintToken" does not emit an event with the exact amount when an external token is used. A contract can emit events when it wants to notify external entities like users, chain explorers, or dApps about changes or conditions in the blockchain. When an event is emitted, it stores the arguments passed in transaction logs. These logs are stored on blockchain and are accessible using address of the contract till the contract is present on the blockchain

The issue exists at the following location:

```
File: /hilter-cgp-solidity/contracts/HilterGateway.sol
465:     function _mintToken(
466:         string memory symbol,
467:         address account,
468:         uint256 amount
469:     ) internal {
470:         address tokenAddress = tokenAddresses(symbol);
471:
472:                         if (tokenAddress == address(0)) revert
TokenDoesNotExist(symbol);
473:
474:         _setTokenDailyMintAmount(symbol, tokenDailyMintAmount(symbol)
+ amount);
475:
476:         if (_getTokenType(symbol) == TokenType.External) {
477:                         bool success = _callERC20Token(tokenAddress,
abi.encodeWithSelector(IERC20.transfer.selector, account, amount));
478:
479:             if (!success) revert MintFailed(symbol);
480:         } else {
481:             IBurnableMintableCappedERC20(tokenAddress).mint(account,
amount);
482:         }
483:     }
```

Which is called by:

```
File: /hilter-cgp-solidity/contracts/HilterGateway.sol
366:
367:        function mintToken(bytes calldata params, bytes32) external
onlySelf {
368:            (string memory symbol, address account, uint256 amount) =
abi.decode(params, (string, address, uint256));
369:
370:          _mintToken(symbol, account, amount);
371:      }
372:
```

The admin command is parsed at the following location:

```
File: /hilter-cgp-solidity/contracts/HilterGateway.sol
262:      function execute(bytes calldata input) external override {
263:          (bytes memory data, bytes memory proof) = abi.decode(input,
(bytes, bytes));
264:
265:                                              bytes32     messageHash     =
ECDSA.toEthSignedMessageHash(keccak256(data));
266:
267:          // TEST auth and getaway separately
268:                                            bool     currentOperators     =
IHilterAuth(AUTH_MODULE).validateProof(messageHash, proof);
269:
270:          uint256 chainId;
271:          bytes32[] memory commandIds;
272:          string[] memory commands;
273:          bytes[] memory params;
274:
275:          try HilterGateway(this)._unpackLegacyCommands(data) returns
(
276:              uint256 chainId_,
277:              bytes32[] memory commandIds_,
278:              string[] memory commands_,
279:              bytes[] memory params_
280:          ) {
```

```
281:                 (chainId, commandIds, commands, params) = (chainId_,
commandIds_, commands_, params_);
282:             } catch {
283:                (chainId, commandIds, commands, params) = abi.decode(data,
(uint256, bytes32[], string[], bytes[]));
284:             }
285:
286:             if (chainId != block.chainid) revert InvalidChainId();
287:
288:             uint256 commandsLength = commandIds.length;
289:
290:              if (commandsLength != commands.length || commandsLength !=
params.length) revert InvalidCommands();
291:
292:             for (uint256 i; i < commandsLength; ++i) {
293:                 bytes32 commandId = commandIds[i];
294:
295:                 if (isCommandExecuted(commandId)) continue; /* Ignore if
duplicate commandId received */
296:
297:                 bytes4 commandSelector;
298:                                                 bytes32    commandHash    =
keccak256(abi.encodePacked(commands[i]));
299:
300:                ...
302:                 } else if (commandHash == SELECTOR_MINT_TOKEN) {
303:                     commandSelector = HilterGateway.mintToken.selector;
310:                ...
312:
```

If the internal token implementation is used, then the mint function of the "MintableCappedERC20" will be called, which indeed will emit an event as part of the open zeppelin ERC20 implementation:

```
File: /hilter-cgp-solidity/contracts/MintableCappedERC20.sol
23:     function mint(address account, uint256 amount) external onlyOwner
{
24:         uint256 capacity = cap;
```

```
25:
26:         _mint(account, amount);
27:
28:         if (capacity == 0) return;
29:
30:         if (totalSupply > capacity) revert CapExceeded();
31:     }
```

File: /hilter-cgp-solidity/contracts/ERC20.sol

```
177:     /** @dev Creates `amount` tokens and assigns them to `account`,
increasing
178:      * the total supply.
179:      *
180:      * Emits a {Transfer} event with `from` set to the zero address.
181:      *
182:      * Requirements:
183:      *
184:      * - `to` cannot be the zero address.
185:      */
186:     function _mint(address account, uint256 amount) internal virtual
{
187:         if (account == address(0)) revert InvalidAccount();
188:
189:         _beforeTokenTransfer(address(0), account, amount);
190:
191:         totalSupply += amount;
192:         balanceOf[account] += amount;
193:         emit Transfer(address(0), account, amount);
194:     }
```

However, in case that an external token implementation is used, it is possible that no event regarding the exact minted amount will be emitted.

On the other hand, an event about the successful execution of the command will be emitted by the "execute" functionality:

```
File: /hilter-cgp-solidity/contracts/HilterGateway.sol
292:         for (uint256 i; i < commandsLength; ++i) {
293:             bytes32 commandId = commandIds[i];
294:
295:             ...
317:
320:                                                 (bool    success,    )    =
address(this).call(abi.encodeWithSelector(commandSelector,      params[i],
commandId));
321:
322:             if (success) emit Executed(commandId);
323:             else _setCommandExecuted(commandId, false);
```

Recommendation

It is recommended to emit an event related to this functionality.

CVSS Score

AV:N/AC:H/PR:N/UI:N/S:U/C:L/I:N/A:N/E:P/RL:X/RC:C/CR:X/IR:X/AR:X/MAV:X/MAC:X/MPR:X/MUI:X/MS:X/MC:X/MI:X/MA:X

5.2.3 Event not emitted in self functionality "collectFees()" at "HilterGasService.sol"

Description	LOW

It was identified that the admin command "collectFees" does not emit an event when the native token is selected. A contract can emit events when it wants to notify external entities like users, chain explorers, or dApps about changes or conditions in the blockchain. When an event is emitted, it stores the arguments passed in transaction logs. These logs are stored on blockchain and are accessible using address of the contract till the contract is present on the blockchain

The issue exists at the following location:

```
File: hilter-cgp-solidity/contracts/gas-service/HilterGasService.sol 122:
function collectFees(address payable receiver, address[] calldata tokens)
external onlyOwner {
123:         for (uint256 i; i < tokens.length; i++) {
124:             address token = tokens[i];
125:
126:             if (token == address(0)) {
127:                 receiver.transfer(address(this).balance);
128:             } else {
129:              uint256 amount = IERC20(token).balanceOf(address(this));
130:                 _safeTransfer(token, receiver, amount);
131:             }
132:         }
133:     }
```

And the "_safeTransfer()" will be:

```
File: /hilter-cgp-solidity/contracts/gas-service/HilterGasService.sol
147:     function _safeTransfer(
148:         address tokenAddress,
149:         address receiver,
150:         uint256 amount
```

```
151:     ) internal {
152:                     (bool success, bytes memory returnData) =
tokenAddress.call(abi.encodeWithSelector(IERC20.transfer.selector,
receiver, amount));
153:         bool transferred = success && (returnData.length == uint256(0)
|| abi.decode(returnData, (bool)));
154:
155:           if (!transferred || tokenAddress.code.length == 0) revert
TransferFailed();
156:     }
157:
```

If the IERC20 token implementation is used, then the "transfer" selector will be called, which will probably emit an event as part of the ERC20 implementation. However, if the ADDRESS_ZERO is used, and the native token is selected, no event will be emitted.

For example, the following test can be used:

```
const destinationChain = 'ethereum';
const destinationAddress = ownerWallet.address;
const payload = defaultAbiCoder.encode(['address', 'address'],
[ownerWallet.address, userWallet.address]);
const symbol = 'USDC';
const amount =0;
const gasToken = testToken.address;
const gasFeeAmount = 0;
const nativeGasFeeAmount =0;
await testToken.connect(userWallet).approve(gasService.address, 0);
await
expect(gasService.connect(ownerWallet).collectFees(ownerWallet.address,
[ADDRESS_ZERO])).to.emit(testToken, 'Transfer');
```

And the output will be:

```
     AssertionError: Expected event "Transfer" to be emitted, but it wasn't
```

Recommendation

It is recommended to emit an event related to this functionality.

CVSS Score

AV:N/AC:H/PR:N/UI:N/S:U/C:L/I:N/A:N/E:P/RL:X/RC:C/CR:X/IR:X/AR:X/MAV:X/MAC:X/MPR:X/MUI:X/MS:X/MC:X/MI:X/MA:X

5.2.4 Event not emitted in self functionality "refund()" at "HilterGasService.sol"

Description	LOW

It was identified that the admin command "refund" does not emit an event, when the native token is selected. A contract can emit events when it wants to notify external entities like users, chain explorers, or dApps about changes or conditions in the blockchain. When an event is emitted, it stores the arguments passed in transaction logs. These logs are stored on blockchain and are accessible using address of the contract till the contract is present on the blockchain

The issue exists at the following location:

```
File: /hilter-cgp-solidity/contracts/gas-service/HilterGasService.sol
135:     function refund(
136:         address payable receiver,
137:         address token,
138:         uint256 amount
139:     ) external onlyOwner {
140:         if (token == address(0)) {
141:             receiver.transfer(amount);
142:         } else {
143:             _safeTransfer(token, receiver, amount);
144:         }
145:     }
```

And the "_safeTransfer()" will be:

```
File: /hilter-cgp-solidity/contracts/gas-service/HilterGasService.sol
147:     function _safeTransfer(
148:         address tokenAddress,
149:         address receiver,
150:         uint256 amount
151:     ) internal {
```

```
152:                     (bool success, bytes memory returnData) =
tokenAddress.call(abi.encodeWithSelector(IERC20.transfer.selector,
receiver, amount));
153:         bool transferred = success && (returnData.length == uint256(0)
|| abi.decode(returnData, (bool)));
154:
155:           if (!transferred || tokenAddress.code.length == 0) revert
TransferFailed();
156:     }
157:
```

If the IERC20 token implementation is used, then the "transfer" selector will be called, which will probably emit an event as part of the ERC20 implementation. However, if the ADDRESS_ZERO is used, and the native token is selected, no event will be emitted.

For example, the following test can be used:

```
const destinationChain = 'ethereum';
const destinationAddress = ownerWallet.address;
const payload = defaultAbiCoder.encode(['address', 'address'],
[ownerWallet.address, userWallet.address]);
const symbol = 'USDC';
const amount = 0;
const gasToken = testToken.address;
const gasFeeAmount = 0;
const nativeGasFeeAmount = 0;
await testToken.connect(userWallet).approve(gasService.address, 0);

 await expect(await
gasService.connect(ownerWallet).refund(userWallet.address, ADDRESS_ZERO,
0x0)).and.to.emit(testToken, 'Transfer').withArgs(gasService.address,
userWallet.address, 0x0);
```

And the output will be:

```
    AssertionError: Expected event "Transfer" to be emitted, but it wasn't
```

Recommendation

It is recommended to emit an event related to this functionality.

CVSS Score

AV:N/AC:H/PR:N/UI:N/S:U/C:L/I:N/A:N/E:P/RL:X/RC:C/CR:X/IR:X/AR:X/MAV:X/MAC:X/MPR:X/MUI:X/MS:X/MC:X/MI:X/MA:X

5.2.5 *Insecure error handling of zero addresses at "ReceiverImplementation.sol" and at "DepositReceiver.sol"*

Description	LOW

The team identified that the "receiveAndSendToken()", "receiveAndSendNative()", "receiveAndUnwrapNative()" functions of ReceiverImplementation and the constructor of DepositReceiver, replace the "refundAddress" with the "msg.sender" when the provided refundAddress is zero. In general, the contract is deployed by the HilterDepositService.sol to act as the recipient address for the cross-chain transfer. When tokens arrive here, it calls the ReceiverImplementation.sol method to forward the tokens to the user, auto-unwrapping if necessary. While the validation of the "refundAddress" parameter is implemented correctly, the error handling is insecure, since the msg.sender might not be able to handle the incoming tokens, especially if instead of an EOA, a contract address is used. In a worst-case scenario, the caller contract logic might lock the incoming funds.

The issue exists in the following locations:

- *contracts/deposit-service/ReceiverImplementation.sol:27:*
  if (refundAddress == address(0)) refundAddress = msg.sender;
- *contracts/deposit-service/ReceiverImplementation.sol:52:*
  if (refundAddress == address(0)) refundAddress = msg.sender;
- *contracts/deposit-service/ReceiverImplementation.sol:77:*
  if (refundAddress == address(0)) refundAddress = msg.sender;
- *contracts/deposit-service/DepositReceiver.sol:25: if (refundAddress == address(0)) refundAddress = msg.sender;*

Since the external functionalities are mainly designed to be used by the upgradable HilterDepositService, the issue is marked as LOW.

Recommendation

It is advisable to verify that the address is not the zero address and then revert the transaction

The functions assert and require can be used to check for conditions and throw an exception if the condition is not met. The control can also be implemented with a simple check:

```
if(refundAddress == address(0)) revert RefundFailed();
```

CVSS Score

AV:N/AC:H/PR:H/UI:N/S:U/C:N/I:L/A:L/E:P/RL:X/RC:C/CR:X/IR:X/AR:X/MAV:X/MAC:X/MPR:X/MUI:X/MS:X/MC:X/MI:X/MA:X

5.2.6 Lack of circuit breaker for emergency stop at "HilterDepositService"

Description	LOW

It was identified that the "HilterDepositService " does not support a circuit breaker control. A circuit breaker, also referred to as an emergency stop, can stop the execution of functions inside the smart contract. A circuit breaker can be triggered manually by trusted parties included in the contract like the contract admin or by using programmatic rules that automatically trigger the circuit breaker when the defined conditions are met. Applying the Emergency Stop pattern to a contract adds a fast and reliable method to halt any sensitive contract functionality as soon as a bug or another security issue is discovered. This leaves enough time to weigh all options and possibly upgrade the contract to fix the security breach.

However, it should be noted that the negative consequence of having an emergency stop mechanism from a user's point of view is, that it adds unpredictable contract behavior.

Recommendation

It is advisable to add a circuit breaker. For example, the following code can be used to set a modifier:

```
bool public contractPaused = false;
function circuitBreaker() public onlyOwner { // onlyOwner can call
    if (contractPaused == false) { contractPaused = true; }
    else { contractPaused = false; }
}
// If the contract is paused, stop the modified function
// Attach this modifier to all public functions
modifier checkIfPaused() {
    require(contractPaused == false);
    _;
}
```

And then:

```
function _execute(
        DepositReceiver depositReceiver,
        address callee,
        uint256 nativeValue,
        bytes memory payload
    ) internal checkIfPaused returns (bool)
```

This approach is similar to openzeppelin pausable contract which can be found in the following URL:

https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/security/Pausable.sol

In both cases, a multisig Owner address must be used to ensure a decentralization strategy.

CVSS Score
AV:N/AC:H/PR:H/UI:N/S:U/C:N/I:N/A:L/E:P/RL:X/RC:C/CR:X/IR:X/AR:X/MAV:X/MAC:X/MPR:X/MUI:X/MS:X/MC:X/MI:X/MA:X

5.2.7 Lack of circuit breaker for emergency stop at "HilterGateway"

Description	LOW

It was identified that the "HilterGateway " does not support a circuit breaker control. A circuit breaker, also referred to as an emergency stop, can stop the execution of functions inside the smart contract. A circuit breaker can be triggered manually by trusted parties included in the contract like the contract admin or by using programmatic rules that automatically trigger the circuit breaker when the defined conditions are met. Applying the Emergency Stop pattern to a contract adds a fast and reliable method to halt any sensitive contract functionality as soon as a bug or another security issue is discovered. This leaves enough time to weigh all options and possibly upgrade the contract to fix the security breach.

Currently, the only way for the admins to halt the transactions is to lower the daily limit to zero. However, this control requires excessive resources, as it will have to be enforced on each affected symbol instead of a global variable, and as a result may not be able to be enforced on time.

```
File: /hilter-cgp-solidity/contracts/HilterGateway.sol
204:         function setTokenDailyMintLimits(string[] calldata symbols,
uint256[] calldata limits) external override onlyAdmin {
205:                     if (symbols.length != limits.length) revert
InvalidSetDailyMintLimitsParams();
206:
207:         for (uint256 i = 0; i < symbols.length; i++) {
208:             string memory symbol = symbols[i];
209:             uint256 limit = limits[i];
210:
211:                     if (tokenAddresses(symbol) == address(0)) revert
TokenDoesNotExist(symbol);
212:
213:             _setTokenDailyMintLimit(symbol, limit);
214:         }
215:     }
```

However, it should be noted that the negative consequence of having an emergency stop mechanism from a user's point of view is, that it adds unpredictable contract behavior.

Recommendation

It is advisable to add a circuit breaker. For example, the following code can be used to set a modifier:

```
bool public contractPaused = false;
function circuitBreaker() public onlyOwner { // onlyOwner can call
    if (contractPaused == false) { contractPaused = true; }
    else { contractPaused = false; }
}
// If the contract is paused, stop the modified function
// Attach this modifier to all public functions
modifier checkIfPaused() {
    require(contractPaused == false);
    _;
}
```

And then:

```
function sendToken(
        string calldata destinationChain,
        string calldata destinationAddress,
        string calldata symbol,
        uint256 amount
    ) external checkIfPaused
```

This approach is similar to openzeppelin pausable contract which can be found in the following URL:

https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/security/Pausable.sol

In both cases, a multisig Owner address must be used to ensure a decentralization strategy.

CVSS Score

AV:N/AC:H/PR:H/UI:N/S:U/C:N/I:N/A:L/E:P/RL:X/RC:C/CR:X/IR:X/AR:X/MAV:X/MAC:X/MPR:X/MUI:X/MS:X/MC:X/MI:X/MA:X

5.2.8 Unvalidated amount in "_mintToken" at "HilterGateway.sol"

Description	LOW

It was identified that internal function "_mintToken" does not ensure that the mint amount is non-zero. Although minting zero tokens is an operation that will neither modify the state of the contract nor produce any results, it will spend the gas the user has provided. Furthermore, it may emit the corresponding event, depending on the ERC20 token's implementation.

The internal function "_mintToken" is called by the admin function "mintToken" and the external function "validateContractCallAndMint", which also do not ensure that the mint amount is non-zero. These external functions are called with operator supplied input data as part of the command execution functionality.

```
File: hilter-cgp-solidity/contracts/HilterGateway.sol
465:     function _mintToken(
466:         string memory symbol,
467:         address account,
468:         uint256 amount
469:     ) internal {
470:         address tokenAddress = tokenAddresses(symbol);
471:
472:                         if   (tokenAddress   ==   address(0))   revert
TokenDoesNotExist(symbol);
473:
474:         _setTokenDailyMintAmount(symbol, tokenDailyMintAmount(symbol)
+ amount);
475:
476:         if (_getTokenType(symbol) == TokenType.External) {
477:                         bool  success  =  _callERC20Token(tokenAddress,
abi.encodeWithSelector(IERC20.transfer.selector, account, amount));
478:
479:             if (!success) revert MintFailed(symbol);
480:         } else {
481:             IBurnableMintableCappedERC20(tokenAddress).mint(account,
amount);
482:         }
```

```
483:     }
```

For example, the following test case will succeed:

```
it('mint tokens with zero amount', async () => {
    const amount = 0;
    const zeroMintData = buildCommandBatch(
        CHAIN_ID,
        [getRandomID()],
        ['mintToken'],
        [getMintCommand(symbol, owner.address, amount)],
    );
    const zeroMintInput = await getSignedMultisigExecuteInput(zeroMintData, operators, operators.slice(0, threshold));
    await expect(gateway.execute(zeroMintInput)).to.emit(gateway, 'Executed');
});
```

And the output will be:

```
 command mintToken
      ✓ mint tokens with zero amount
```

Recommendation

It is recommended to verify that the mint amount is greater than zero.

The functions "assert" and "require" can be used to check for conditions and throw an exception if the condition is not met. The control can also be implemented with a simple check:

```
if (amount == 0) revert InvalidAmount();
```

CVSS Score

AV:N/AC:H/PR:H/UI:N/S:U/C:N/I:L/A:N/E:P/RL:X/RC:C/CR:X/IR:X/AR:X/MAV:X/MAC:X/MPR:X/MUI:X/MS:X/MC:X/MI:X/MA:X

5.2.9 Unvalidated amount in "burn" and "burnFrom" at "BurnableMintableCappedERC20.sol"

Description	LOW

It was found that the "amount" in external function "burnFrom()" and the "balanceOf[account]" in external "burn()" is not validated to be non-zero. Although burning zero tokens is an operation that will neither modify the state of the contract nor produce any results, it will spend the gas the user has provided. Furthermore, it is possible to burn zero tokens from any account and emit the corresponding event, since and the default allowance for all accounts is zero and the corresponding check will succeed.

The issue exists at the following function:

```
File: hilter-cgp-solidity/contracts/BurnableMintableCappedERC20.sol 39:
function burnFrom(address account, uint256 amount) external onlyOwner {
40:         uint256 _allowance = allowance[account][msg.sender];
41:         if (_allowance != type(uint256).max) {
42:             _approve(account, msg.sender, _allowance - amount);
43:         }
44:         _burn(account, amount);
45:     }
```

It should be noted that when the functions are called from the HilterGateway, only the "burn()" can be exploited, as the "_burnTokenFrom()" which calls the "burnFrom()" already contains a such security control as it can be seen below:

```
File: hilter-cgp-solidity/contracts/HilterGateway.sol
485:     function _burnTokenFrom(
486:         address sender,
487:         string memory symbol,
488:         uint256 amount
489:     ) internal {
490:         address tokenAddress = tokenAddresses(symbol);
491:
```

```
492:                         if (tokenAddress == address(0)) revert
TokenDoesNotExist(symbol);
493:         if (amount == 0) revert InvalidAmount();
494:
```

The following test case can be used to replicate this issue:

```
const burnAmount = 0;
await token.transfer(depositHandlerAddress, burnAmount);
const dataFirstBurn = buildCommandBatch(CHAIN_ID, [getRandomID()],
['burnToken'], [getBurnCommand(symbol, salt)]);
const firstInput = await getSignedMultisigExecuteInput(dataFirstBurn,
operators, operators.slice(0, threshold));
await expect(gateway.execute(firstInput)).to.emit(token,
'Transfer').withArgs(depositHandlerAddress, ADDRESS_ZERO, burnAmount);
```

And the output will be:

```
    command burnToken
      ✓ able to burn zero tokens
```

Recommendation

It is advisable to verify that the amount is not zero.

The functions assert and require can be used to check for conditions and throw an exception if the condition is not met. The control can also be implemented with a simple check.

In burn():

```
 if (balanceOf[account] == 0) revert InvalidAmount();
```

In burnFrom():

```
if (amount == 0) revert InvalidAmount();
```

CVSS Score
AV:N/AC:H/PR:H/UI:N/S:U/C:N/I:L/A:N/E:P/RL:X/RC:C/CR:X/IR:X/AR:X/MAV:X/MAC:X/MPR:X/MUI:X/MS:X/MC:X/MI:X/MA:X

5.2.10 Unvalidated amount in "refund" at "HilterGasService.sol"

Description	LOW

It was found that the "amount" in the external function "refund" is not validated to be non-zero. Although requesting a refund of zero tokens is an operation that will neither modify the state of the contract nor produce any results, it will spend the gas the user has provided. Furthermore, it may be possible to transfer zero tokens from the Gas Service and emit the corresponding event even though the user is not eligible for a refund, depending on the ERC20 token's implementation.

File: hilter-cgp-solidity/contracts/gas-service/HilterGasService.sol

```
135:     function refund(
136:         address payable receiver,
137:         address token,
138:         uint256 amount
139:     ) external onlyOwner {
140:         if (token == address(0)) {
141:             receiver.transfer(amount);
142:         } else {
143:             _safeTransfer(token, receiver, amount);
144:         }
145:     }
```

The following test case can be used to replicate the issue:

```
await expect(await
gasService.connect(ownerWallet).refund(userWallet.address,
testToken.address, 0x0))
.and.to.emit(testToken, 'Transfer')
.withArgs(gasService.address, userWallet.address, 0x0);
```

And the output would be:

```
gas receiver
     ✓ refund zero amount
```

Recommendation

It is advisable to verify that the amount is not zero.

The functions assert and require can be used to check for conditions and throw an exception if the condition is not met. The control can also be implemented with a simple check:

```
if (amount == 0) revert InvalidAmount();
```

CVSS Score

AV:N/AC:H/PR:H/UI:N/S:U/C:N/I:L/A:N/E:P/RL:X/RC:C/CR:X/IR:X/AR:X/MAV:X/MAC:X/MPR:X/MUI:X/MS:X/MC:X/MI:X/MA:X

5.2.11 No multisig protection in "util/upgradable.sol"

Description	LOW

The team identified that the admin role (owner) of the "util/upgradable.sol" contract is not protected with multisig. Smart contracts have privileged roles that are responsible to perform operations such as minting, pausing, and upgrading, which are necessary in the lifecycle of a project. The best practice for securing admin accounts is to use a multisig. A multisig is a contract that can execute actions, as long as a predefined number of trusted members agree upon it. A multisig has a number of owners (N) and requires some of them (M) to approve a transaction. This configuration is referred to as M of N.

In the specific case, the admin role (owner) of the contract is responsible for transferring the ownership of the contract and upgrading the contract:

```
File: hilter-cgp-solidity/contracts/util/Upgradable.sol
25:       function transferOwnership(address newOwner) external virtual
onlyOwner {
...
32:     }
41:     function upgrade(
42:         address newImplementation,
43:         bytes32 newImplementationCodeHash,
44:         bytes calldata params
45:     ) external override onlyOwner {
...
59:     }
```

However, the team identified that the owner is verified only by comparing the msg.sender with a stored address in the storage slot:

```
File: hilter-cgp-solidity/contracts/util/Upgradable.sol
11:                   bytes32    internal    constant    _OWNER_SLOT    =
0x02016836a56b71f0d02689e69e326f4f4c1b9057164ef592671cf0d37c8040c0;
12:
13:     modifier onlyOwner() {
```

```
14:         if (owner() != msg.sender) revert NotOwner();
15:         _;
16:     }
17:
18:     function owner() public view returns (address owner_) {
19:         // solhint-disable-next-line no-inline-assembly
20:         assembly {
21:             owner_ := sload(_OWNER_SLOT)
22:         }
23:     }
```

File: hilter-cgp-solidity/contracts/deposit-service/HilterDepositService.sol

```
11:
12: // This should be owned by the microservice that is paying for gas.
13: contract HilterDepositService is Upgradable, IHilterDepositService {
```

Recommendation

It is advisable to protect the owner functions of the upgradable contract with multisig.

CVSS Score

AV:N/AC:H/PR:H/UI:N/S:U/C:N/I:L/A:N/E:P/RL:X/RC:C/CR:X/IR:X/AR:X/MAV:X/MAC:X/MPR:X/MUI:X/MS:X/MC:X/MI:X/MA:X

5.2.12 Unvalidated amount in "payGasForContractCall()", "payGasForContractCallWithToken()", and the "addGas()" functions at "HilterGasService.sol"

Description	LOW

It was found that the "amount" in the external function "_safeTransferFrom" is not validated to be non-zero. The function is currently used by the "payGasForContractCall()", "payGasForContractCallWithToken()", and the "addGas()" external functions. Although transferring zero tokens is an operation that will neither modify the state of the contract nor produce any results, it will spend the gas the user has provided. Furthermore, it may be possible to transfer zero tokens from any account and emit the corresponding event, depending on the ERC20 token's implementation.

The issue exists at the following function:

```
File: hilter-cgp-solidity/contracts/gas-service/HilterGasService.sol
158:     function _safeTransferFrom(
159:         address tokenAddress,
160:         address from,
161:         uint256 amount
162:     ) internal {
163:         (bool success, bytes memory returnData) = tokenAddress.call(
164:            abi.encodeWithSelector(IERC20.transferFrom.selector, from,
address(this), amount)
165:         );
166:        bool transferred = success && (returnData.length == uint256(0)
|| abi.decode(returnData, (bool)));
167:
168:           if (!transferred || tokenAddress.code.length == 0) revert
TransferFailed();
169:     }
```

Currently, the "_safeTransferFrom" is called from a number of external functions. The "payGasForContractCall()":

```
File: /hilter-cgp-solidity/contracts/gas-service/HilterGasService.sol
11:     // This is called on the source chain before calling the gateway
to execute a remote contract.
12:     function payGasForContractCall(
13:         address sender,
14:         string calldata destinationChain,
15:         string calldata destinationAddress,
16:         bytes calldata payload,
17:         address gasToken,
18:         uint256 gasFeeAmount,
19:         address refundAddress
20:     ) external override {
21:         _safeTransferFrom(gasToken, msg.sender, gasFeeAmount);
```

The "payGasForContractCallWithToken()":

```
File: /hilter-cgp-solidity/contracts/gas-service/HilterGasService.sol
35:     function payGasForContractCallWithToken(
36:         address sender,
37:         string calldata destinationChain,
38:         string calldata destinationAddress,
39:         bytes calldata payload,
40:         string memory symbol,
41:         uint256 amount,
42:         address gasToken,
43:         uint256 gasFeeAmount,
44:         address refundAddress
45:     ) external override {
46:         {
47:             _safeTransferFrom(gasToken, msg.sender, gasFeeAmount);
48:         }
```

And the "addGas()":

```
File: /hilter-cgp-solidity/contracts/gas-service/HilterGasService.sol
100:     function addGas(
101:         bytes32 txHash,
102:         uint256 logIndex,
103:         address gasToken,
104:         uint256 gasFeeAmount,
105:         address refundAddress
106:     ) external override {
107:         _safeTransferFrom(gasToken, msg.sender, gasFeeAmount);
108:
109:             emit GasAdded(txHash, logIndex, gasToken, gasFeeAmount,
refundAddress);
110:     }
```

For example, the following test case will succeed:

```
    it('zero gas is added', async () => {
        const   txHash   =   keccak256(defaultAbiCoder.encode(['string'],
['random tx hash']));
        const logIndex = 13;
        const gasToken = testToken.address;
        const gasFeeAmount = 0;
        const nativeGasFeeAmount = parseEther('1.0');

        await   testToken.connect(userWallet).approve(gasService.address,
1e6);


        await        expect(gasService.connect(userWallet).addGas(txHash,
logIndex, gasToken, gasFeeAmount, userWallet.address))
            .to.emit(gasService, 'GasAdded')
            .withArgs(txHash,    logIndex,    gasToken,    gasFeeAmount,
userWallet.address)
            .and.to.emit(testToken, 'Transfer')
            .withArgs(userWallet.address,             gasService.address,
gasFeeAmount);
```

```
});
```

And the output will be:

```
  HilterGasService
    ✓ zero gas is added
```

Recommendation

It is advisable to verify that the amount is not zero. Since all functions use the "_safeTransferFrom" internal function, the check can be performed there.

The functions assert and require can be used to check for conditions and throw an exception if the condition is not met. The control can also be implemented with a simple check:

```
if (amount == 0) revert InvalidAmount();
```

CVSS Score

AV:N/AC:H/PR:H/UI:N/S:U/C:N/I:L/A:N/E:P/RL:X/RC:C/CR:X/IR:X/AR:X/MAV:X/MAC:X/MPR:X/MUI:X/MS:X/MC:X/MI:X/MA:X

5.2.13 Unvalidated amount in "collectfees" at "HilterGasService.sol"

Description	LOW

It was identified that the function "collectFees()" does not validate the amount parameter. Although transfering zero tokens is an operation that will neither modify the state of the contract nor produce any results, it will spend the gas the user has provided. Furthermore, it may emit the corresponding event, depending on the ERC20 token's implementation.

The issue exists at the following function:

```
File: hilter-cgp-solidity/contracts/gas-service/HilterGasService.sol 122:
function collectFees(address payable receiver, address[] calldata tokens)
external onlyOwner {
123:         for (uint256 i; i < tokens.length; i++) {
124:             address token = tokens[i];
125:
126:             if (token == address(0)) {
127:                 receiver.transfer(address(this).balance);
128:             } else {
129:              uint256 amount = IERC20(token).balanceOf(address(this));
130:                 _safeTransfer(token, receiver, amount);
131:             }
132:         }
133:     }
```

which will call either the "receiver.transfer()" or the "_safeTransfer()". And the "_safeTransfer()":

```
File: hilter-cgp-solidity/contracts/gas-service/HilterGasService.sol
147:     function _safeTransfer(
148:         address tokenAddress,
149:         address receiver,
150:         uint256 amount
151:     ) internal {
```

```
152:                     (bool success, bytes memory returnData) =
tokenAddress.call(abi.encodeWithSelector(IERC20.transfer.selector,
receiver, amount));
153:         bool transferred = success && (returnData.length == uint256(0)
|| abi.decode(returnData, (bool)));
154:
155:             if (!transferred || tokenAddress.code.length == 0) revert
TransferFailed();
156:     }
```

For example, the following test case will succeed:

```
        it('collect zero fees', async () => {
            const destinationChain = 'ethereum';
            const destinationAddress = ownerWallet.address;
            const payload = defaultAbiCoder.encode(['address', 'address'],
[ownerWallet.address, userWallet.address]);
            const symbol = 'USDC';
            const amount = 0;
            const gasToken = testToken.address;
            const gasFeeAmount = 0;
            const nativeGasFeeAmount = 0;
            await
testToken.connect(userWallet).approve(gasService.address, 0);
            await                                                expect(await
gasService.connect(ownerWallet).collectFees(ownerWallet.address,        [
testToken.address]))
                .to.changeEtherBalance(ownerWallet, nativeGasFeeAmount)
                .and.to.emit(testToken, 'Transfer')
                .withArgs(gasService.address,        ownerWallet.address,
gasFeeAmount);
        });
```

And the output will be:

```
    gas receiver
      ✓ collect zero fees
```

Recommendation

It is recommended to verify that the amount is greater than zero.

The functions "assert" and "require" can be used to check for conditions and throw an exception if the condition is not met. The control can also be implemented with a simple check:

```
if (amount == 0) revert InvalidAmount();
```

CVSS Score

AV:N/AC:H/PR:H/UI:N/S:U/C:N/I:L/A:N/E:P/RL:X/RC:C/CR:X/IR:X/AR:X/MAV:X/MAC:X/MPR:X/MUI:X/MS:X/MC:X/MI:X/MA:X

5.2.14 Unvalidated address "receiver" in "collectFees" at "HilterGasService.sol"

Description	LOW

It was found that the "receiver" in the external function "collectFees" is not validated to not be the zero address. Transferring a number of tokens to the zero address is equivalent to burning that number of tokens.

The issue exists at the following function:

```
File: hilter-cgp-solidity/contracts/gas-service/HilterGasService.sol 122:
function collectFees(address payable receiver, address[] calldata tokens)
external onlyOwner {
123:         for (uint256 i; i < tokens.length; i++) {
124:             address token = tokens[i];
125:
126:             if (token == address(0)) {
127:                 receiver.transfer(address(this).balance);
128:             } else {
129:              uint256 amount = IERC20(token).balanceOf(address(this));
130:                 _safeTransfer(token, receiver, amount);
131:             }
132:         }
133:     }
```

For example, the following test case can be used to replicate the issue:

```
await    expect(gasService.connect(ownerWallet).collectFees(ADDRESS_ZERO,
[ADDRESS_ZERO]));
```

And the output will be:

```
    gas receiver
      ✓ collectfees to zero address
```

Recommendation

It is advisable to verify that the address is not the zero address.

The functions assert and require can be used to check for conditions and throw an exception if the condition is not met. The control can also be implemented with a simple check:

```
if (receiver == address(0)) revert InvalidReceiver();
```

CVSS Score

AV:N/AC:H/PR:H/UI:N/S:U/C:N/I:L/A:N/E:P/RL:X/RC:C/CR:X/IR:X/AR:X/MAV:X/MAC:X/MPR:X/MUI:X/MS:X/MC:X/MI:X/MA:X

5.2.15 Unvalidated address "receiver" in "refund" at "HilterGasService.sol"

Description	LOW

It was found that the "receiver" in the external function "refund" is not validated to not be the zero address. Transfering an amount of tokens to the zero address is equivalent to burning that amount of tokens.

The issue exists at the following function:

```
File: hilter-cgp-solidity/contracts/gas-service/HilterGasService.sol
135:     function refund(
136:         address payable receiver,
137:         address token,
138:         uint256 amount
139:     ) external onlyOwner {
140:         if (token == address(0)) {
141:             receiver.transfer(amount);
142:         } else {
143:             _safeTransfer(token, receiver, amount);
144:         }
145:     }
```

The following test case can be used to replicate this issue:

```
await expect(gasService.connect(ownerWallet).refund(ADDRESS_ZERO, ADDRESS_ZERO, gasFeeAmount));
```

And the output will be:

```
    gas receiver
      ✓ refund to zero address
```

Recommendation

It is advisable to verify that the address is not the zero address.

The functions assert and require can be used to check for conditions and throw an exception if the condition is not met. The control can also be implemented with a simple check:

```
if (receiver == address(0)) revert InvalidReceiver();
```

CVSS Score

AV:N/AC:H/PR:H/UI:N/S:U/C:N/I:L/A:N/E:P/RL:X/RC:C/CR:X/IR:X/AR:X/MAV:X/MAC:X/MPR:X/MUI:X/MS:X/MC:X/MI:X/MA:X

5.2.16 Unvalidated address "recipient" in "withdrawNative" at "HilterDepositService.sol"

Description	LOW

It was found that the "recipient" in the external function "withdrawNative" is not validated to not be the zero address. Transferring a number of tokens to the zero address is equivalent to burning that number of tokens.

The issue exists at the following function:

```
File: /hilter-cgp-solidity/contracts/deposit-service/HilterDepositService.sol
115:     function withdrawNative(bytes32 salt, address payable recipient) external {
116:         address token = wrappedToken();
117:         DepositReceiver depositReceiver = new DepositReceiver{
118:           salt: keccak256(abi.encode(PREFIX_DEPOSIT_WITHDRAW_NATIVE, salt, recipient))
119:         }();
120:                                                 uint256 amount = IERC20(token).balanceOf(address(depositReceiver));
121:
122:         if (amount == 0) revert NothingDeposited();
123:
124:                         if (!_execute(depositReceiver, token, 0, abi.encodeWithSelector(IWETH9.withdraw.selector, amount))) revert UnwrapFailed();
125:
126:          // NOTE: `depositReceiver` must always be destroyed in the same runtime context that it is deployed.
127:         depositReceiver.destroy(recipient);
128:     }
```

The following test case can be used to replicate this issue:

```
 If('unwrap native currency to zero address', async () => {
        const recipient = userWallet.address;
        const salt = formatBytes32String(1);
        const amount = 1e6;
        const              depositAddress              =              await
depositService.depositAddressForWithdrawNative(salt, ADDRESS_ZERO);
        await          token.connect(ownerWallet).transfer(depositAddress,
amount);
        await       expect(await       depositService.withdrawNative(salt,
ADDRESS_ZERO));
    });
```

And the output will be:

```
    gas receiver
      ✓ unwrap native currency to zero address
```

Recommendation

It is advisable to verify that the address is not the zero address.

The functions assert and require can be used to check for conditions and throw an exception:

```
if(recipient == address(0)) revert UnwrapFailed();
```

CVSS Score

AV:N/AC:H/PR:H/UI:N/S:U/C:N/I:L/A:N/E:P/RL:X/RC:C/CR:X/IR:X/AR:X/MAV:X/MAC:X/MPR:X/MUI:X/MS:X/MC:X/MI:X/MA:X

5.3 Informational Findings

5.3.1 Ownership can be transferred to same owner at "Ownable.sol"

Description	INFO

It was identified that the "transferOwnership" functionality does not validate if the new owner is the same with the existing owner. Currently, the "transferOwnership" function allows the current owner to transfer control of an Ownable contract to a newOwner.

The issue exists at the following location:

```
File: /hilter-cgp-solidity/contracts/Ownable.sol
21:       function transferOwnership(address newOwner) external virtual
onlyOwner {
22:         if (newOwner == address(0)) revert InvalidOwner();
23:
24:         emit OwnershipTransferred(owner, newOwner);
25:         owner = newOwner;
26:     }
```

A user, who is the owner of the specific contract, could use this function in order to transfer the ownership again back to them, creating an event of this transaction.

It should be noted that the same logic is also implemented in the *Ownable.sol* contract from Open Zeppelin

Recommendation

It is advisable to verify that the newOwner is different than the current owner.

The functions assert and require can be used to check for conditions and throw an exception if the condition is not met. The control can also be implemented with a simple check:

```
if(newOwner == owner) revert
```

CVSS Score

AV:N/AC:L/PR:N/UI:N/S:U/C:N/I:N/A:N/E:P/RL:X/RC:C/CR:X/IR:X/AR:X/MAV:X/MAC:X/MPR:X/MUI:X/MS:X/MC:X/MI:X/MA:X

5.3.2 Lack of circuit breaker for emergency stop at "HilterGasService"

Description	INFO

It was identified that the "HilterGasService" does not support a circuit breaker control. A circuit breaker, also referred to as an emergency stop, can stop the execution of functions inside the smart contract. A circuit breaker can be triggered manually by trusted parties included in the contract like the contract admin or by using programmatic rules that automatically trigger the circuit breaker when the defined conditions are met. Applying the Emergency Stop pattern to a contract adds a fast and reliable method to halt any sensitive contract functionality as soon as a bug or another security issue is discovered. This leaves enough time to weigh all options and possibly upgrade the contract in order to fix the security breach.

Recommendation

It is advisable to add a circuit breaker. For example, the following code can be used to set a modifer:

```
bool public contractPaused = false;
function circuitBreaker() public onlyOwner { // onlyOwner can call
    if (contractPaused == false) { contractPaused = true; }
    else { contractPaused = false; }
}
// If the contract is paused, stop the modified function
// Attach this modifier to all public functions
modifier checkIfPaused() {
    require(contractPaused == false);
    _;
}
```

```
function _safeTransferFrom(
        address tokenAddress,
        address from,
        uint256 amount
    ) internal checkIfPaused returns (bool)
```

This approach is similar to openzeppelin pausable contract which can be found in the following URL:

https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/security/Pausable.sol

In both cases, a multisig Owner address must be used to ensure a decentralization strategy.

CVSS Score

AV:N/AC:H/PR:N/UI:N/S:U/C:N/I:N/A:N/E:P/RL:X/RC:C/CR:X/IR:X/AR:X/MAV:X/MAC:X/MPR:X/MUI:X/MS:X/MC:X/MI:X/MA:X

5.3.3 Excessive loop iterations allowed in "setAdmins" at "AdminMultisigBase.sol"

Description	INFO

It was identified that the internal function "_setAdmins", which is only called by the administrator -only "setup" function in "HilterGateway .sol", contains a potentially costly loop. Computational power on blockchain environments is paid , thus reducing the computational steps required to complete an operation is not only a matter of optimization but also cost efficiency. Loops are a great example of costly operations : as many elements an array has, more iterations will be required to complete the loop.

Excessive loop iterations may exhaust all available gas. For example, if an attacker is able to influence the element array's length, then they will be able to cause a denial of service, preventing the execution to jump out of the loop.

In the specific case, the function "_setAdmins" iterates over the "adminAddresses" array which is decoded from the provided argument and is of unspecified length:

```
File: hilter-cgp-solidity/contracts/AdminMultisigBase.sol
144:     function _setAdmins(
145:         uint256 adminEpoch,
146:         address[] memory accounts,
147:         uint256 threshold
148:     ) internal {
149:         uint256 adminLength = accounts.length;
...
158:         for (uint256 i; i < adminLength; ++i) {
```

The function is called by:

```
File: /hilter-cgp-solidity/contracts/HilterGateway.sol
241:     function setup(bytes calldata params) external override {
242:             // Prevent setup from being called on a non-proxy (the
implementation).
```

```
243:         if (implementation() == address(0)) revert NotProxy();
244:
245:         (address[] memory adminAddresses, uint256 newAdminThreshold,
bytes memory newOperatorsData) = abi.decode(
246:             params,
247:             (address[], uint256, bytes)
248:         );
249:
250:           // NOTE: Admin epoch is incremented to easily invalidate
current admin-related state.
251:         uint256 newAdminEpoch = _adminEpoch() + uint256(1);
252:         _setAdminEpoch(newAdminEpoch);
253:         _setAdmins(newAdminEpoch, adminAddresses, newAdminThreshold);
254:
...
260:     }
```

Recommendation

It is advisable to refactor the logic to not require to set all administrators in one transaction (if required), or to insert an upper limit that will allow the operation to be performed without failing due to insufficient gas.

If it is absolutely necessary to loop over an array of unknown size, the function should be able to execute the operation in multiple blocks and in multiple transactions. In that case, it will be required to maintain the extra state of how many iterations have already been performed in order to continue from that point in the next function call.

CVSS Score

AV:N/AC:L/PR:N/UI:N/S:U/C:N/I:N/A:N/E:P/RL:X/RC:C/CR:X/IR:X/AR:X/MAV:X/MAC:X/MPR:X/MUI:X/MS:X/MC:X/MI:X/MA:X

5.3.4 Excessive loop iterations allowed in "admins" at "HilterGateway.sol"

Description	INFO

It was identified that the external function "admins", which can be called by any user, contains a potentially costly loop. Computational power on blockchain environments is paid, thus reducing the computational steps required to complete an operation is not only a matter of optimization but also cost efficiency. Loops are a great example of costly operations: as many elements an array has, more iterations will be required to complete the loop.

Excessive loop iterations exhaust all available gas. For example, if an attacker is able to influence the element array's length, then they will be able to cause a denial of service, preventing the execution to jump out of the loop.

In the specific case, the function "admins()" which returns an array with all the available admins, will iterate based on the result of the "_getAdminCount()" functionality:

```
File: hilter-cgp-solidity/contracts/HilterGateway.sol
190:     /// @dev Returns the array of admins within a given `adminEpoch`.
191:       function admins(uint256 epoch) external view override returns
(address[] memory results) {
192:         uint256 adminCount = _getAdminCount(epoch);
193:         results = new address[](adminCount);
194:
195:         for (uint256 i; i < adminCount; ++i) {
196:             results[i] = _getAdmin(epoch, i);
197:         }
198:     }
```

And the "_getAdminCount()":

```
File: hilter-cgp-solidity-contracts/AdminMultisigBase.sol
100:     function _getAdminCount(uint256 adminEpoch) internal view returns
(uint256) {
101:         return getUint(_getAdminCountKey(adminEpoch));
```

```
102:     }
```

which eventually will retrieve it from the storage:

```
File: hilter-cgp-solidity/contracts/AdminMultisigBase.sol
64:         function _getAdminCountKey(uint256 adminEpoch) internal pure
returns (bytes32) {
65:                 return keccak256(abi.encodePacked(PREFIX_ADMIN_COUNT,
adminEpoch));
66:     }
67:
```

This could previously be configured at:

```
File: hilter-cgp-solidity/contracts/AdminMultisigBase.sol
140:     function _setAdminCount(uint256 adminEpoch, uint256 adminCount)
internal {
141:         _setUint(_getAdminCountKey(adminEpoch), adminCount);
142:     }
143:
```

Which is used at:

```
File: hilter-cgp-solidity/contracts/AdminMultisigBase.sol
144:     function _setAdmins(
145:         uint256 adminEpoch,
146:         address[] memory accounts,
147:         uint256 threshold
148:     ) internal {
149:         uint256 adminLength = accounts.length;
150:
151:         if (adminLength < threshold) revert InvalidAdmins();
152:
153:         if (threshold == uint256(0)) revert InvalidAdminThreshold();
154:
```

```
155:         _setAdminThreshold(adminEpoch, threshold);
156:         _setAdminCount(adminEpoch, adminLength);
157:
158:         for (uint256 i; i < adminLength; ++i) {
159:             address account = accounts[i];
160:
161:             // Check that the account wasn't already set as an admin
for this epoch.
162:                         if (_isAdmin(adminEpoch, account)) revert
DuplicateAdmin(account);
163:
164:             if (account == address(0)) revert InvalidAdmins();
165:
166:           // Set this account as the i-th admin in this epoch (needed
to we can clear topic votes in `onlyAdmin`).
167:             _setAdmin(adminEpoch, i, account);
168:             _setIsAdmin(adminEpoch, account, true);
169:         }
```

And this is configured at the "setup" functionality:

```
File: hilter-cgp-solidity/contracts/HilterGateway.sol
241:     function setup(bytes calldata params) external override {
242:             // Prevent setup from being called on a non-proxy (the
implementation).
243:         if (implementation() == address(0)) revert NotProxy();
244:
245:         (address[] memory adminAddresses, uint256 newAdminThreshold,
bytes memory newOperatorsData) = abi.decode(
246:             params,
247:             (address[], uint256, bytes)
248:         );
249:
250:           // NOTE: Admin epoch is incremented to easily invalidate
current admin-related state.
251:         uint256 newAdminEpoch = _adminEpoch() + uint256(1);
252:         _setAdminEpoch(newAdminEpoch);
253:         _setAdmins(newAdminEpoch, adminAddresses, newAdminThreshold);
```

```
254:
```

Recommendation

It is advisable to refactor the logic to return the admins in multiple transactions, or to insert an upper limit that will allow the operation to be performed without failing due to insufficient gas.

CVSS Score

AV:N/AC:L/PR:N/UI:N/S:U/C:N/I:N/A:N/E:P/RL:X/RC:C/CR:X/IR:X/AR:X/MAV:X/MAC:X/MPR:X/MUI:X/MS:X/MC:X/MI:X/MA:X

5.3.5 Excessive loop iterations allowed in "collectFees" at "HilterGasService.sol"

Description	INFO

It was identified that the external function "collectFees", which can be called only by the contract's owner, contains a potentially costly loop. Computational power on blockchain environments is paid, thus reducing the computational steps required to complete an operation is not only a matter of optimization but also cost efficiency. Loops are a great example of costly operations: as many elements an array has, more iterations will be required to complete the loop.

In the specific case, the function "collectFees" iterates over the "tokens" array, which is provided as argument and is of unspecified length:

```
File: hilter-cgp-solidity/contracts/gas-service/HilterGasService.sol 122:
function collectFees(address payable receiver, address[] calldata tokens)
external onlyOwner {
123:         for (uint256 i; i < tokens.length; i++) {
124:             address token = tokens[i];
125:
126:             if (token == address(0)) {
127:                 receiver.transfer(address(this).balance);
128:             } else {
129:              uint256 amount = IERC20(token).balanceOf(address(this));
130:                 _safeTransfer(token, receiver, amount);
131:             }
132:         }
133:     }
```

Recommendation

It is advisable to refactor the logic to not require to collect all the fees in one transaction.

Alternatively, If it is absolutely necessary to loop over an array of unknown size, the function should plan for it to potentially take multiple blocks and therefore require multiple transactions. In that case, it will be required to maintain the extra state of how many iterations have already been performed in order to continue from that point in the next function call. However, this may cause additional issues if other functions are executed while waiting for the next iteration of this function to be executed.

CVSS Score

AV:N/AC:L/PR:N/UI:N/S:U/C:N/I:N/A:N/E:P/RL:X/RC:C/CR:X/IR:X/AR:X/MAV:X/MAC:X/MPR:X/MUI:X/MS:X/MC:X/MI:X/MA:X

5.3.6 Excessive loop iterations allowed in "execute" at "HilterGateway.sol"

Description	INFO

It was identified that the external function "execute", which can be called only by the gateway operators, contains a potentially costly loop. Computational power on blockchain environments is paid, thus reducing the computational steps required to complete an operation is not only a matter of optimization but also cost efficiency. Loops are a great example of costly operations: as many elements an array has, more iterations will be required to complete the loop.

Excessive loop iterations may exhaust all available gas. For example, if an attacker can influence the element array's length, then they will be able to cause a denial of service, preventing the execution to jump out of the loop.

In the specific case, the function "execute" iterates over the "commands" array which is decoded from the provided arguments and is of unspecified length:

```
File: hilter-cgp-solidity/contracts/HilterGateway.sol
262:     function execute(bytes calldata input) external override {269:
270:         ….
291:
292:         for (uint256 i; i < commandsLength; ++i) {
293:             bytes32 commandId = commandIds[i];
294:
295:             if (isCommandExecuted(commandId)) continue; /* Ignore if
duplicate commandId received */
296:
…
324:         }
325:     }
```

Recommendation

It is advisable to refactor the logic to perform the operation in multiple transactions, or to insert an upper limit that will allow the operation to be performed without failing due to insufficient gas.

CVSS Score

AV:N/AC:L/PR:N/UI:N/S:U/C:N/I:N/A:N/E:P/RL:X/RC:C/CR:X/IR:X/AR:X/MAV:X/MAC:X/MPR:X/MUI:X/MS:X/MC:X/MI:X/MA:X

5.3.7 No reentrancy protection in "execute" at "DepositReceiver.sol"

Description	INFO

It was identified that the "execute" function of the DepositReceiver.sol is protected from Reentrancy attacks. This type of attack can occur when a contract sends ether to an unknown address . An attacker can carefully construct a contract at an external address that contains malicious code in the fallback function . Thus , when a contract sends ether to this address , it will invoke the malicious code. Typically, the malicious code executes a function on the vulnerable contract , performing operations not expected by the developer.

```
File: hilter-cgp-solidity/contracts/deposit-service/DepositReceiver.sol
20:     function execute(
21:         address callee,
22:         uint256 value,
23:         bytes calldata data
24:     ) external onlyOwner returns (bool success, bytes memory returnData)
{
25:         if (callee.code.length == 0) revert NotContract();
26:
27:         (success, returnData) = callee.call{ value: value }(data);
28:     }
```

Recommendation

It is advisable to also use the "ReentrancyGuard" as an added layer of security.

CVSS Score

AV:N/AC:H/PR:H/UI:N/S:U/C:N/I:N/A:N/E:P/RL:X/RC:C/CR:X/IR:X/AR:X/MAV:X/MAC:X/MPR:X/MUI:X/MS:X/MC:X/MI:X/MA:X

5.3.8 Floating pragma in multiple interfaces at "contracts/interfaces/" folder

Description	INFO

It was found that many interfaces of smart contracts are using a floating pragma. In Solidity programming, multiple APIs only be supported in some specific versions. In each contract, the pragma keyword is used to enable certain compiler features or checks. If a contract does not specify a compiler version, developers might encounter compile errors in the future code reuse because of the version gap.

The issue exists at:

- *contracts/interfaces/IOwnable.sol:3:pragma solidity ^0.8.9;*
- *contracts/interfaces/IMintableCappedERC20.sol:3:pragma solidity ^0.8.9;*
- *contracts/interfaces/IERC20.sol:3:pragma solidity ^0.8.9;*
- *contracts/interfaces/IUpgradable.sol:3:pragma solidity ^0.8.9;*
- *contracts/interfaces/IERC20BurnFrom.sol:3:pragma solidity ^0.8.9;*
- *contracts/interfaces/IERC20Permit.sol:3:pragma solidity ^0.8.9;*
- *contracts/interfaces/IHilterExecutable.sol:3:pragma solidity ^0.8.9;*
- *contracts/interfaces/IWETH9.sol:3:pragma solidity ^0.8.9;*
- *contracts/interfaces/IERC20Burn.sol:3:pragma solidity ^0.8.9;*
- *contracts/interfaces/IHilterForecallable.sol:3:pragma solidity ^0.8.9;*
- *contracts/interfaces/IBurnableMintableCappedERC20.sol:3:pragma solidity ^0.8.9;*
- *contracts/interfaces/IDepositServiceBase.sol:3:pragma solidity ^0.8.9;*
- *contracts/interfaces/IHilterGateway.sol:3:pragma solidity ^0.8.9;*
- *contracts/interfaces/IHilterDepositService.sol:3:pragma solidity ^0.8.9;*
- *contracts/interfaces/IHilterAuthWeighted.sol:3:pragma solidity ^0.8.9;*
- *contracts/interfaces/IHilterGasService.sol:3:pragma solidity ^0.8.9;*
- *contracts/interfaces/IHilterAuth.sol:3:pragma solidity ^0.8.9;*
- *contracts/interfaces/ITokenDeployer.sol:3:pragma solidity ^0.8.9*;

Recommendation

Source files should be annotated with a pragma version to reject compilation with previous or future compiler versions that might introduce incompatible changes.

It is recommended to avoid using the "^" directive to avoid using nightly builds,

CVSS Score

AV:N/AC:H/PR:H/UI:N/S:U/C:N/I:N/A:N/E:P/RL:X/RC:C/CR:X/IR:X/AR:X/MAV:X/MAC:X/MPR:X/MUI:X/MS:X/MC:X/MI:X/MA:X

5.3.9 Setup functionality can be circumvented during contract upgrade at "/contracts/util/Upgradable.sol"

Description	INFO

It was identified that the "upgradable" contracts allow the upgrade to take place without calling the "setup" functionality. In general, upgradable contracts are not able to use constructors to store data due to the proxy design. As a result, a well-protected initialization functionality such as the "setup()" function is used to perform the required operations. However, in the specific case, it was found that the upgrade can take place without calling this functionality, by just not providing any parameters. The initialization phase of an upgradeable smart contract is one of the most important phases. If not properly handled, it can compromise a smart contract with perfect business logic implementation.

The issue exists at:

```
File: /hilter-cgp-solidity-4.3.0/contracts/util/Upgradable.sol
49:
50:         if (params.length > 0) {
51:             // solhint-disable-next-line avoid-low-level-calls
52:                                                 (bool    success,    )    =
newImplementation.delegatecall(abi.encodeWithSelector(this.setup.selector
, params));
53:
54:             if (!success) revert SetupFailed();
55: }
```

Recommendation

It is advisable to always call the "setup()" initialization functionality by default.

CVSS Score

AV:N/AC:H/PR:H/UI:N/S:U/C:N/I:N/A:N/E:P/RL:X/RC:C/CR:X/IR:X/AR:X/MAV:X/MAC:X/MPR:X/MUI:X/MS:X/MC:X/MI:X/MA:X

6 Retest Results

6.1 Retest of Medium Severity Findings

All MEDIUM-risk findings has been fixed.

6.2 Retest of Low Severity Findings

All LOW-risk vulnerabilities were found to be sufficiently mitigated, since the affected functionality has been fixed or removed.

6.3 Retest of Informational Findings

All INFORMATIONAL findings has been fixed.

References & Applicable Documents

Ref.	Title	Version
N/A	*N/A*	*N/A*

Document History

Revision	Description	Changes Made By	Date
0.2	*Initial Draft*	*Chaintroopers*	*June 21 2025*
1.0	*First Version*	*Chaintroopers*	*July 15 , 2025*
1.1	*Added retest results* *Added v4.3.0 results*	*Chaintroopers*	*August 3, 2025*