// Security Assessment

01.20.2025 - 01.21.2025

Hashgraph: Wrapped HBAR Smart Contract Project

Hashgraph

HALBORN

Hashgraph: Wrapped HBAR Smart Contract Project - Hashgraph

Prepared by: HALBORN

Last Updated 03/12/2025

Date of Engagement by: January 20th, 2025 - January 21st, 2025

Summary

100% OF ALL REPORTED FINDINGS HAVE BEEN ADDRESSED

ALL FINDINGS	CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
4	0	0	0	1	3

TABLE OF CONTENTS

1. Introduction
2. Assessment summary
3. Test approach and methodology
4. Caveats
5. Risk methodology
6. Scope
7. Assessment summary & findings overview
8. Findings & Tech Details
   - 8.1 Use of transfer instead of call
   - 8.2 Use custom errors to give more information about failures
   - 8.3 Floating pragma
   - 8.4 Consider using named mappings
9. Automated Testing

1. Introduction

Hashgraph engaged Halborn to conduct a security assessment on their smart contracts beginning on January 20th, 2025 and ending on January 21st, 2025. The security assessment was scoped to the smart contracts provided to Halborn. Commit hashes and further details can be found in the Scope section of this report.

The Hashgraph codebase in scope consist of one smart contract:

- WHBAR.sol is a wrapper contract for the native HBAR token.

2. Assessment Summary

Halborn was provided 2 days for the engagement and assigned 1 full-time security engineer to review the security of the smart contracts in scope. The engineer is a blockchain and smart contract security expert with advanced smart contract hacking skills, and deep knowledge of multiple blockchain protocols.

The purpose of the assessment is to:

- Identify potential security issues within the smart contracts.
- Ensure that smart contract functionality operates as intended.

In summary, Halborn identified some improvements to reduce the likelihood and impact of risks, which were completely addressed by the Hashgraph team. The main ones were the following:

- Consider using the 'call' method for transferring native funds, as it is more flexible and handles the increase in gas cost associated with the transfer.
- Consider locking the pragma version to the same version used during development and testing.
- Consider using named mappings.
- Consider using custom errors to give more information about failures.

3. Test Approach And Methodology

`Halborn` performed a combination of manual review of the code and automated security testing to balance efficiency, timeliness, 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 smart contracts 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, purpose and use of the platform.
- Smart contract manual code review and walkthrough to identify any logic issue.
- Thorough assessment of safety and usage of critical Solidity variables and functions in scope that could lead to arithmetic related vulnerabilities.
- Local testing with custom scripts (`Foundry`).
- Fork testing against main networks (`Foundry`).
- Static analysis of security for scoped contract, and imported functions(`Slither`).

4. Caveats

The `Hashgraph team` introduced additional changes in order to mitigate an edge case on Hedera, where it is possible to make a network `CryptoTransfer call` which won't invoke the smart contract and thus the amount returned from the `totalSupply()` function will be different than the actual amount of `WHBAR`. Additionally sending `HBAR` to the `WHBAR.sol` contract by selfdestructing a contract shouldn't increase the amount returned by the `totalSupply()` function. The changes in the following PR:https://github.com/hashgraph/hedera-smart-contracts/pull/1256/files were thoroughly examined by `Halborn`, and it was determined that they don't introduce any new vulnerabilities.

5. 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.

5.1 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) Specific (AO:S)	1 0.2
Attack Cost (AC)	Low (AC:L) Medium (AC:M) High (AC:H)	1 0.67 0.33
Attack Complexity (AX)	Low (AX:L) Medium (AX:M) High (AX:H)	1 0.67 0.33

Exploitability E is calculated using the following formula:

$$E = \prod m_e$$

5.2 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) Low (I:L) Medium (I:M) High (I:H) Critical (I:C)	0 0.25 0.5 0.75 1
Integrity (I)	None (I:N) Low (I:L) Medium (I:M) High (I:H) Critical (I:C)	0 0.25 0.5 0.75 1
Availability (A)	None (A:N) Low (A:L) Medium (A:M) High (A:H) Critical (A:C)	0 0.25 0.5 0.75 1
Deposit (D)	None (D:N) Low (D:L) Medium (D:M) High (D:H) Critical (D:C)	0 0.25 0.5 0.75 1
Yield (Y)	None (Y:N) Low (Y:L) Medium (Y:M) High (Y:H) Critical (Y:C)	0 0.25 0.5 0.75 1

Impact I is calculated using the following formula:

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

5.3 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.

METRICS:

SEVERITY COEFFICIENT (C)	COEFFICIENT VALUE	NUMERICAL VALUE
Reversibility (r)	None (R:N) Partial (R:P) Full (R:F)	1 0.5 0.25
Scope (s)	Changed (S:C) Unchanged (S:U)	1.25 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

SEVERITY	SCORE VALUE RANGE
Informational	0 - 1.9

6. SCOPE

FILES AND REPOSITORY

(a) Repository: hedera-smart-contracts

(b) Assessed Commit ID: 41c5b50

(c) Items in scope:

- contracts/wrapped-tokens/WHBAR.sol

Out-of-Scope:

Out-of-Scope: New features/implementations after the remediation commit IDs.

7. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW
0	0	0	1

INFORMATIONAL
3

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
HAL-01 - USE OF TRANSFER INSTEAD OF CALL	LOW	SOLVED - 01/01/2025
HAL-02 - USE CUSTOM ERRORS TO GIVE MORE INFORMATION ABOUT FAILURES	INFORMATIONAL	SOLVED - 01/01/2025

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
HAL-03 - FLOATING PRAGMA	INFORMATIONAL	SOLVED - 01/01/2025
HAL-04 - CONSIDER USING NAMED MAPPINGS	INFORMATIONAL	SOLVED - 01/01/2025

8. FINDINGS & TECH DETAILS

8.1 (HAL-01) USE OF TRANSFER INSTEAD OF CALL

// LOW

Description

When a user wants to withdraw the native tokens he deposited into the `WHBAR.sol` contract, he has to call the `withdraw()` function:

```
    function withdraw(uint wad) public {
        require(balanceOf[msg.sender] >= wad);

        balanceOf[msg.sender] -= wad;
        payable(msg.sender).transfer(wad);

        emit Withdrawal(msg.sender, wad);
    }
```

However, as can be seen from the above code snippet, `transfer()` is used. The `transfer()` method forwards 2300 gas stipend, and this can be dangerous for two reasons:

1. Gas costs of EVM instructions may change significantly during hard forks, which may previously assumed fixed gas costs. EIP 1884 as an example, broke several existing smart contracts due to a cost increase of the SLOAD instruction.
2. If the recipient is a contract or a multisig safe, with a `receive`/`fallback` function which requires >2300 gas, e.g safes that execute extra logic in the `receive`/`fallback` function, the transfer function will always fail for them due to out of gas errors.

Given this is the only way depositors can withdraw the native tokens they have deposited, or they have been transferred from someone, else this may render the critical withdraw functionality obsolete.

BVSS

AO:A/AC:L/AX:M/R:N/S:U/C:N/A:N/I:N/D:M/Y:N (3.4)

Recommendation

Consider using the `call()` method for transferring funds, as it is more flexible and handles the increase in gas cost associated with the transfer.

Remediation Comment

SOLVED: The **Hashgraph team** has followed the recommendation and successfully resolved the issue.

References

hashgraph/hedera-smart-contracts/contracts/wrapped-tokens/WHBAR.sol#L35

8.2 (HAL-02) USE CUSTOM ERRORS TO GIVE MORE INFORMATION ABOUT FAILURES

// INFORMATIONAL

Description

Throughout the code, simple require statements are used without strings to give more information about failures (e.g., `revert("Insufficient funds.");`). It is recommended to use custom errors, to give more information as to why a function failed, this helps with testing and maintaining the code base. Utilizing the `Error()` syntax is an optimization strategy that can significantly reduce gas costs. Hard-coded strings, stored on the blockchain, increase the size and cost of deploying and executing contracts.

The `Error()` syntax allows for the definition of reusable, parameterized custom errors, leading to a more efficient use of storage and reduced gas consumption. This approach not only optimizes gas usage during deployment and interaction with the contract but also enhances code maintainability and readability by providing clearer, context-specific error information.

BVSS

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

Recommendation

It is recommended to give more information about failures and use custom errors, which can be done following the logic below:

1. Standard require statement (to be replaced):

```
require(condition, "Condition not met");
```

2. Declare the error definition to state

```
error ConditionNotMet();
```

3. As currently is not possible to use custom errors in combination with `require` statements, the standard syntax is:

```
if (!condition) revert ConditionNotMet();
```

More information about this topic in Official Solidity Documentation.

Remediation Comment

SOLVED: The **Hashgraph team** has followed the recommendation and successfully resolved the issue.

References

hashgraph/hedera-smart-contracts/contracts/wrapped-tokens/WHBAR.sol#L32

8.3 (HAL-03) FLOATING PRAGMA

// INFORMATIONAL

Description

The contract in scope currently uses floating pragma versions `^0.8.22` which means that the code can be compiled by any compiler version that is greater than or equal to `0.8.22`, and less than `0.9.0`.

However, it is recommended that 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.

In this aspect, it is crucial to select the appropriate EVM version when it's intended to deploy the contracts on networks other than the Ethereum mainnet, which may not support these opcodes. Failure to do so could lead to unsuccessful contract deployments or transaction execution issues.

BVSS

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

Recommendation

Lock the pragma version to the same version used during development and testing. Additionally, make sure to specify the target EVM version when using Solidity versions from `0.8.20` and above if deploying to chains that may not support newly introduced opcodes. Additionally, it is crucial to stay informed about the opcode support of different chains to ensure smooth deployment and compatibility.

Remediation Comment

SOLVED: The **Hashgraph team** has followed the recommendation and successfully resolved the issue.

References

hashgraph/hedera-smart-contracts/contracts/wrapped-tokens/WHBAR.sol#L2

8.4 (HAL-04) CONSIDER USING NAMED MAPPINGS

// INFORMATIONAL

Description

The project is using a Solidity version greater than `0.8.18`, which supports named mappings. Using named mappings can improve the readability and maintainability of the code by making the purpose of each mapping clearer. This practice helps developers and auditors understand the mappings' intent more easily.

BVSS

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

Recommendation

Consider refactoring the mappings to use named arguments, which will enhance code readability and make the purpose of each mapping more explicit.

For example, in the `WHBAR.sol` contract instead of declaring:

```
mapping(address => uint) public balanceOf;
```

It could be declared as:

```
mapping(address user => uint balance) public balanceOf;
```

Remediation Comment

SOLVED: The **Hashgraph team** has followed the recommendation and successfully resolved the issue.

References

hashgraph/hedera-smart-contracts/contracts/wrapped-tokens/WHBAR.sol#L14

9. AUTOMATED TESTING

Static Analysis Report

Description

Halborn used automated testing techniques to enhance the coverage of certain areas of the smart contracts in scope. Among the tools used was Slither, a Solidity static analysis framework. After Halborn verified the smart contracts in the repository and was able to compile them correctly into their abis and binary format, Slither was run against the contract. This tool can statically verify mathematical relationships between Solidity variables to detect invalid or inconsistent usage of the contracts' APIs across the entire code-base.
The security team assessed all findings identified by the Slither software, however, most of the findings are not included in the below results for the sake of report readability.

Output

The findings obtained as a result of the Slither scan were reviewed, and the majority were not included in the report because they were determined as false positives.

Halborn strongly recommends conducting a follow-up assessment of the project either within six months or immediately following any material changes to the codebase, whichever comes first. This approach is crucial for maintaining the project's integrity and addressing potential vulnerabilities introduced by code modifications.