// Public Smart Contract Security Assessment 04.08.2024 - 04.10.2024

Axelar Bridge Adapter

Moonwell

HALBORN

Axelar Bridge Adapter - Moonwell

Prepared by: HALBORN

Last Updated 04/23/2024

Date of Engagement by: April 8th, 2024 - April 10th, 2024

Summary

0% OF ALL REPORTED FINDINGS HAVE BEEN ADDRESSED

ALL FINDINGS	CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
0	0	0	0	0	0

TABLE OF CONTENTS

1. Introduction

`Moonwell` engaged `Halborn` to conduct a security assessment on their `AxelarBridgeAdapter.sol` smart contract **beginning on April 8th and ending on April 10th**. The security assessment was scoped to the smart contract provided in the moonwell-fi/moonwell-contracts-v2 GitHub repository. Commit hash and further details can be found in the Scope section of this report.

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 contract in scope. The engineer is a blockchain and smart contract security expert with advanced penetration testing and 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 contract `AxelarBridgeAdapter.sol`.
- Ensure that smart contract functionality operates as intended.

In summary, Halborn has not identified any security risk inherent to the contract under analysis, given the engagement scope.

3. Test Approach And Methodology

Halborn performed a combination of manual and automated security testing to balance efficiency, timeliness, practicality, and accuracy in regard to the scope of this assessment. While manual testing is recommended to uncover flaws in logic, process, and implementation; automated testing techniques help enhance coverage of the code and can quickly identify items that do not follow the security best practices. The following phases and associated tools were used during the assessment:

- Research into architecture and purpose.
- Smart contract manual code review and walkthrough.
- Graphing out functionality and contract logic/connectivity/functions (`solgraph`).
- Manual assessment of use and safety for the critical Solidity variables and functions in scope to identify any arithmetic-related vulnerability classes.
- Manual testing by custom scripts.
- Static Analysis of security for scoped contract, and imported functions (`slither`).
- Testnet deployment (`Foundry`).

4. Manual Testing

The contract in scope was thoroughly and manually analyzed for potential vulnerabilities and bugs, as well as known optimizations and best practices when developing Smart Contracts in Solidity.

While no major vulnerabilities were found within the scope and time frame provided, it's always important to highlight good practices that were identified during the assessment, which contribute positively to the security maturity of the contracts in-scope, such as:

- Thorough documentation using NatSpec.

- The use of `unchecked` blocks in for loops extends gas optimization.

- The usage of `Ownable2Step` pattern is considered a good security practice and mitigates this risk by introducing a two-step process for ownership transfer. The current owner initiates the transfer by proposing a new owner, but the transfer only completes when the proposed new owner accepts it.

- Contract calls are validated in the `execute` function through the call to `gateway.validateContractCall`, this helps preventing unallowed actions passing through the execution mechanism.

These security practices are applied industry-wide and should be preserved in future implementations and developments.

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:

EXPLOITABILIY METRIC (m_e)	METRIC VALUE	NUMERICAL VALUE
Attack Origin (AO)	Arbitrary (AO:A) Specific (AO:S)	1 0.2

EXPLOITABILIY METRIC (m_e)	METRIC VALUE	NUMERICAL VALUE
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

SEVERITY	SCORE VALUE RANGE
Low	2 - 4.4
Informational	0 - 1.9

6. SCOPE

FILES AND REPOSITORY

(a) Repository: moonwell-contracts-v2

(b) Assessed Commit ID: 0b9c111

(c) Contracts in scope:

- src/xWELL/AxelarBridgeAdapter.sol

Out-of-Scope:

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

7. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
0	0	0	0	0

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE

8. FINDINGS & TECH DETAILS

9. REVIEW NOTES

Removed style/optimization issues from final report as requested by client.

10. AUTOMATED TESTING

Introduction

Halborn used automated testing techniques to enhance the coverage of certain areas of the smart contract in scope. Among the tools used was Slither, a Solidity static analysis framework. After Halborn verified the smart contract in the repository and was able to compile it correctly into their ABI 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 contract's API across the entire code-base.
The security team assessed all findings identified by the Slither software, and findings with severity `Information` and `Optimization` are also included in the below results.

```
INFO:Detectors:
Reentrancy in xERC20BridgeAdapter._bridgeIn(uint256,address,uint256) (src/xWELL/xERC20BridgeAdapter.sol#83-91):
        External calls:
        - xERC20.mint(user,amount) (src/xWELL/xERC20BridgeAdapter.sol#88)
        Event emitted after the call(s):
        - BridgedIn(chainId,user,amount) (src/xWELL/xERC20BridgeAdapter.sol#90)
Reentrancy in AxelarBridgeAdapter._bridgeOut(address,uint256,uint256,address) (src/xWELL/AxelarBridgeAdapter.sol#286-319):
        External calls:
        - _burnTokens(user,amount) (src/xWEL
                - xERC20.burn(user,amount) (src/xWELL/xERC20BridgeAdapter.sol#75)
        - gasService.payNativeGasForContractCall{value: msg.value}(address(this),chainIdToAxelarId[dstChainId],address(this).toString(),payload,user) (src/xWELL/AxelarBridgeAdapter.sol#303-309)
        - gateway.callContract(chainIdToAxelarId[dstChainId],address(this).toString(),payload) (src/xWELL/AxelarBridgeAdapter.sol#312-316)
        External calls sending eth:
        - gasService.payNativeGasForContractCall{value: msg.value}(address(this),chainIdToAxelarId[dstChainId],address(this).toString(),payload,user) (src/xWELL/AxelarBridgeAdapter.sol#303-309)
        Event emitted after the call(s):
        - BridgedOut(dstChainId,user,to,amount) (src/xWELL/AxelarBridgeAdapter.sol#318)
Reentrancy in AxelarBridgeAdapter.execute(bytes32,string,string,bytes) (src/xWELL/AxelarBridgeAdapter.sol#332-365):
        External calls:
        - require(bool,string)(gateway.validateContractCall(commandId,sourceChain,sourceAddress,payloadHash),AxelarBridgeAdapter: call not approved by gateway) (src/xWELL/AxelarBridgeAdapter.sol#349-357)
        - _bridgeIn(axelarIdToChainId[sourceChain],user,amount) (src/xWELL/AxelarBridgeAdapter.sol#364)
                - xERC20.mint(user,amount) (src/xWELL/xERC20BridgeAdapter.sol#88)
        Event emitted after the call(s):
        - BridgedIn(chainId,user,amount) (src/xWELL/xERC20BridgeAdapter.sol#90)
                - _bridgeIn(axelarIdToChainId[sourceChain],user,amount) (src/xWELL/AxelarBridgeAdapter.sol#364)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#reentrancy-vulnerabilities-3
INFO:Detectors:
AxelarBridgeAdapter._removeChainId(AxelarBridgeAdapter.ChainIds) (src/xWELL/AxelarBridgeAdapter.sol#259-273) has costly operations inside a loop:
        - delete chainIdToAxelarId[chainids.chainid] (src/xWELL/AxelarBridgeAdapter.sol#269)
AxelarBridgeAdapter._removeChainId(AxelarBridgeAdapter.ChainIds) (src/xWELL/AxelarBridgeAdapter.sol#259-273) has costly operations inside a loop:
        - delete axelarIdToChainId[chainids.axelarid] (src/xWELL/AxelarBridgeAdapter.sol#270)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#costly-operations-inside-a-loop
INFO:Detectors:
Pragma version0.8.19 (src/xWELL/axelarInterfaces/AddressString.sol#3) necessitates a version too recent to be trusted. Consider deploying with 0.8.18.
Pragma version^0.8.0 (src/xWELL/axelarInterfaces/IAxelarGasService.sol#3) allows old versions
Pragma version^0.8.0 (src/xWELL/axelarInterfaces/IAxelarGateway.sol#18) allows old versions
Pragma version^0.8.0 (src/xWELL/axelarInterfaces/IGovernable.sol#18) allows old versions
Pragma version0.8.19 (src/xWELL/interfaces/IXERC20.sol#1) necessitates a version too recent to be trusted. Consider deploying with 0.8.18.
Pragma version0.8.19 (src/xWELL/xERC20BridgeAdapter.sol#1) necessitates a version too recent to be trusted. Consider deploying with 0.8.18.
Pragma version0.8.19 (src/xWELL/AxelarBridgeAdapter.sol#1) necessitates a version too recent to be trusted. Consider deploying with 0.8.18.
solc-0.8.19 is not recommended for deployment
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#incorrect-versions-of-solidity
INFO:Detectors:
Contract xERC20BridgeAdapter (src/xWELL/xERC20BridgeAdapter.sol#10-104) is not in CapWords
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#conformance-to-solidity-naming-conventions
INFO:Slither:src/xWELL/AxelarBridgeAdapter.sol analyzed (17 contracts with 93 detectors), 14 result(s) found
```


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