Moonwell Finance - XWell Token & Rate-Limiting

Smart Contract Security Assessment

Prepared by: **Halborn**

Date of Engagement: **November 15th, 2023 - January 1st, 2024**

Visit: **Halborn.com**

DOCUMENT REVISION HISTORY

VERSION	MODIFICATION	DATE
0.1	Document Creation	11/21/2023
0.2	Document Updates	11/27/2023
0.3	Draft Review	11/27/2023
1.0	Remediation Plan	11/27/2023
1.1	Remediation Plan Review	11/28/2023
2.0	Document Updates	01/01/2024
2.1	Document Updates Review	01/02/2024

CONTACTS

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

EXECUTIVE OVERVIEW

1.1 INTRODUCTION

Moonwell Finance engaged Halborn to conduct a security assessment on their smart contracts beginning on November 15th, 2023 and ending on January 1st, 2024. The security assessment was scoped to the smart contracts provided to the Halborn team.

1.2 ASSESSMENT SUMMARY

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

The purpose of this assessment is to:

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

In summary, Halborn identified some security risks that were successfully addressed by the Moonwell Finance team.

1.3 TEST APPROACH & METHODOLOGY

Halborn performed a combination of manual and automated security testing to balance efficiency, timeliness, practicality, and accuracy in regard to the scope of this 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)

2. RISK METHODOLOGY

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

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

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

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

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

2.1 EXPLOITABILITY

Attack Origin (AO):

Captures whether the attack requires compromising a specific account.

Attack Cost (AC):

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

Attack Complexity (AX):

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

Metrics:

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

Exploitability E is calculated using the following formula:

$$E = \prod m_e$$

2.2 IMPACT

Confidentiality (C):

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

Integrity (I):

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

Availability (A):

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

Deposit (D):

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

Yield (Y):

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

Metrics:

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

Impact I is calculated using the following formula:

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

2.3 SEVERITY COEFFICIENT

Reversibility (R):

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

Scope (S):

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

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

Severity Coefficient C is obtained by the following product:

$$C = rs$$

The Vulnerability Severity Score S is obtained by:

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

The score is rounded up to 1 decimal places.

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

2.4 SCOPE

1. IN-SCOPE TREE & COMMIT :

The security assessment was scoped to the following contract:

- zelt

COMMIT ID : 447e7ab0eccfc1b6614714e3c63b8fdefae98076

- src/lib/RateLimitedMidpointLibrary.sol.
- src/lib/RateLimitMidpointCommonLibrary.sol.

REMEDIATION COMMIT IDs :

- 4e66e79485448ff31890a06618902135cace12d1

  ----

2. IN-SCOPE TREE (XWELL) & COMMIT :

- WormholeBridge Adapter contract
- Pull Request 88
- Pull Request 81
- Pull Request 97

- src/xWELL/xWELL.sol.
- src/xWELL/XERC20Lockbox.sol.
- src/xWELL/xERC20BridgeAdapter.sol.

- src/xWELL/WormholeBridgeAdapter.sol.
- src/xWELL/xWELLRouter.sol.

REMEDIATION COMMIT IDs :

- 326357fc1fc1c5fdd83116ff9fb1f7cbf093d597
- 45f7b230ab68e97c77e7f06abeb930dc6149d825

3. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
0	1	0	0	2

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
(HAL-01) MISSING ERC20PERMIT INIT CALL IN INITIALIZE FUNCTION OF CONTRACT XWELL	High (7.0)	SOLVED - 11/28/2023
(HAL-02) REDUNDANT EVENTS ON THE RateLimitedMidpointLibrary LIBRARY	Informational (1.6)	SOLVED - 11/24/2023
(HAL-03) REDUNDANT USAGE OF SAFECAST IN XERC20LOCKBOX CONTRACT	Informational (1.6)	SOLVED - 11/28/2023

FINDINGS & TECH DETAILS

4.1 (HAL-01) MISSING ERC20PERMIT INIT CALL IN INITIALIZE FUNCTION OF CONTRACT XWELL - HIGH (7.0)

Description:

In the Solidity smart contract xWELL, which inherits from ERC20VotesUpgradeable, there seems to be an omission in the initialization process. The initialize function, intended to set up the contract's initial state, does not include a call to **__ERC20Permit_-init. **This function is critical for initializing the ERC20 Permit feature, which is part of the ERC20VotesUpgradeable contract. The ERC20 Permit feature allows for gasless transactions by enabling users to sign approvals for token transfers with their private keys. Not calling** __ERC20Permit_init** means that this functionality will not be properly set up in the xWELL contract.

Code Location:

xWELL.sol#L17

Listing 1

```
    /// @dev on token's native chain, the lockbox must have its
↳ bufferCap set to uint112 max
    /// @notice initialize the xWELL token
    /// @param tokenName The name of the token
    /// @param tokenSymbol The symbol of the token
    /// @param tokenOwner The owner of the token, Temporal
↳ Governor on Base, Timelock on Moonbeam
    /// @param newRateLimits The rate limits for the token
    function initialize(
        string memory tokenName,
        string memory tokenSymbol,
        address tokenOwner,
        MintLimits.RateLimitMidPointInfo[] memory newRateLimits,
        uint128 newPauseDuration,
        address newPauseGuardian
    ) external initializer {
        require(
            newPauseDuration <= MAX_PAUSE_DURATION,
            "xWELL: pause duration too long"
        );
        __ERC20_init(tokenName, tokenSymbol);
        __Ownable_init();
        _addLimits(newRateLimits);

        /// pausing
        __Pausable_init(); /// not really needed, but seems like
↳ good form
        _grantGuardian(newPauseGuardian); /// set the pause
↳ guardian
        _updatePauseDuration(newPauseDuration);

        transferOwnership(tokenOwner);
    }
```

Proof Of Concept:

- Without __ERC20Permit_init, the ERC20 Permit feature is non-functional.
- Users are unable to perform gasless transactions, a key feature expected from ERC20VotesUpgradeable.

BVSS:

AO:A/AC:L/AX:L/C:N/I:C/A:M/D:N/Y:N/R:P/S:C (7.0)

Recommendation:

Modify the initialize function to include a call to __ERC20Permit_init. This function should be called with appropriate arguments (usually the name of the token) to properly initialize the ERC20 Permit feature.

Remediation Plan:

SOLVED: The Moonwell Finance team solved the issue by calling the __ERC20Permit_init function.

Commit ID: 326357fc1fc1c5fdd83116ff9fb1f7cbf093d597

4.2 (HAL-02) REDUNDANT EVENTS ON THE RateLimitedMidpointLibrary LIBRARY - INFORMATIONAL (1.6)

Description:

In the library, specifically within the RateLimitedMidpointLibrary, there are redundant event definitions for RateLimitPerSecondUpdate and BufferCapUpdate. These events are already defined in the RateLimitMidpointCommonLibrary. Additionally, having duplicate event definitions increases the size of the compiled contract unnecessarily, which can have implications on deployment costs and efficiency.

Code Location:

RateLimitedMidpointLibrary.sol#L18C1-L25C7

Listing 2

```
...
    /// @notice event emitted when buffer cap is updated
    event BufferCapUpdate(uint256 oldBufferCap, uint256
↳ newBufferCap);

    /// @notice event emitted when rate limit per second is
↳ updated
    event RateLimitPerSecondUpdate(
        uint256 oldRateLimitPerSecond,
        uint256 newRateLimitPerSecond
    );
...
```

BVSS:

AO:A/AC:L/AX:L/C:N/I:M/A:N/D:N/Y:N/R:F/S:C (1.6)

Recommendation:

Consider removing redundant events.

Remediation Plan:

SOLVED: The Moonwell Finance team solved the issue by removing redundant events.

Commit ID: 4e66e79485448ff31890a06618902135cace12d1

4.3 (HAL-03) REDUNDANT USAGE OF SAFECAST IN XERC20LOCKBOX CONTRACT - INFORMATIONAL (1.6)

Description:

In the XERC20Lockbox, there is an import statement for SafeCast from OpenZeppelin's contracts library, but upon reviewing the contract's code, it appears that SafeCast is not being used. SafeCast is typically employed for safely casting between different numeric types, ensuring that the cast does not cause unintended overflows or underflows. However, in this contract, all operations involving uint256 types do not utilize any casting that would require SafeCast.

Code Location:

XERC20Lockbox.sol#L15

Listing 3

```
contract XERC20Lockbox is IXERC20Lockbox {
    using SafeERC20 for IERC20;
    using SafeCast for uint256;
}
```

BVSS:

AO:A/AC:L/AX:L/C:N/I:M/A:N/D:N/Y:N/R:F/S:C (1.6)

Recommendation:

Since SafeCast is not utilized in the contract, it's recommended to remove the import statement. This helps in reducing the contract's compilation size and improves readability by eliminating unnecessary code.

Remediation Plan:

SOLVED: The Moonwell Finance team solved the issue by removing the redundant SafeCast import.

Commit ID: 45f7b230ab68e97c77e7f06abeb930dc6149d825

REVIEW SUMMARY

This audit report provides an in-depth analysis of the xWELL token, integral to the Moonwell protocol. The report focuses on assessing the security, integrity, and overall functionality of the token. A noteworthy aspect of this audit is the extensive testing regimen undertaken by the Moonwell team. The rigor and comprehensiveness of these tests significantly contribute to our assessment and findings.

Testing Methodologies Employed by the Moonwell team:

- Unit Testing: The Moonwell team performed detailed unit testing on the xWELL token. Each component was isolated and rigorously tested for functionality, ensuring that all individual modules operated correctly.
- Fuzz Testing: Fuzz testing, conducted by the Moonwell team, played a crucial role in uncovering potential vulnerabilities. By subjecting the xWELL token to unpredictable and varied inputs, the team effectively identified and addressed hidden bugs.
- Integration Testing: The Moonwell team's integration testing ensured that the xWELL token's components worked cohesively. This testing phase was crucial in validating the integrated functionality of the entire system.
- Invariant Testing: Critical invariants, unique to the Moonwell ecosystem, were thoroughly tested by the Moonwell team. This process verified the consistency and reliability of essential business rules and conditions within the system.

The Moonwell team's proactive and comprehensive approach to testing has established a strong foundation for the xWELL token's security and operational integrity. Our audit findings affirm the effectiveness of these testing efforts. We recommend maintaining these rigorous testing standards in ongoing and future developments.

AUTOMATED TESTING

6.1 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 contracts. 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.

Results:

- No major issues found by Slither.

THANK YOU FOR CHOOSING

// HALBORN