// Security Assessment

06.02.2025 - 06.03.2025

XLockup Contract

Usual

HALBORN

XLockup Contract - Usual

Prepared by: HALBORN

Last Updated 06/10/2025

Date of Engagement: June 2nd, 2025 - June 3rd, 2025

Summary

100% ⓘ OF ALL REPORTED FINDINGS HAVE BEEN ADDRESSED

ALL FINDINGS	CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
2	0	1	0	0	1

TABLE OF CONTENTS

1. Introduction

`Usual` engaged Halborn to conduct a security assessment of their smart contracts from June 2nd to June 3rd, 2025. The assessment scope was limited to the smart contracts provided to the Halborn team. Commit hashes and additional details are available in the Scope section of this report.

2. Assessment Summary

The Halborn team dedicated two days to this engagement, assigning one full-time security engineer to evaluate the security of the smart contracts.

The assigned security engineer is an expert in blockchain and smart contract security, with advanced skills in penetration testing, smart contract exploitation, and extensive knowledge of multiple blockchain protocols.

The objectives of this assessment were to:

- Verify that the smart contract functions operate as intended.
- Identify potential security vulnerabilities within the smart contracts.

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

- `Add lock duration validation when releasing and re-locking.`
- `Implement zero value validation.`

3. Test Approach And Methodology

Halborn employed a combination of manual, semi-automated, and automated security testing to balance efficiency, timeliness, practicality, and accuracy within the scope of this assessment. Manual testing is essential for uncovering flaws in logic, process, and implementation, while automated techniques enhance code coverage and quickly identify deviations from security best practices. The following phases and tools were utilized during the assessment:

- Research into the architecture and intended functionality.
- Manual code review and walkthrough of the smart contracts.
- Manual assessment of critical Solidity variables and functions to identify potential vulnerability classes.
- Manual testing using custom scripts.
- Static security analysis of the scoped contracts and imported functions using `Slither`.
- Local deployment and testing using `Foundry`.

4. 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.

4.1 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

EXPLOITABILITY METRIC (M_E)	METRIC VALUE	NUMERICAL VALUE
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:

$$E = \prod m_e$$

4.2 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

IMPACT METRIC (M_I)	METRIC VALUE	NUMERICAL VALUE
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:

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

4.3 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
Informational	0 - 1.9

5. SCOPE

FILES AND REPOSITORY

(a) Repository: core-protocol

(b) Assessed Commit ID: f497c2e

(c) Items in scope:

- src/vaults/UsualXLockup.sol
- src/vaults/UsualX.sol:depositAndLock()
- src/vaults/UsualX.sol:depositAndLockWithPermit()

Out-of-Scope: Third party dependencies and economic attacks.

REMEDIATION COMMIT ID:

- 8bb2dc0

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

6. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW
0	1	0	0

INFORMATIONAL
1

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
LOCK DURATION VALIDATION BYPASS	HIGH	SOLVED - 06/02/2025

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
NO MINIMUM LOCK AMOUNT VALIDATION	INFORMATIONAL	ACKNOWLEDGED - 06/10/2025

7. FINDINGS & TECH DETAILS

7.1 LOCK DURATION VALIDATION BYPASS

// HIGH

Description

The `releaseAndLock()` function in `UsualXLockup.sol` bypasses lock duration validation by invoking `_lock()` directly without verifying whether the specified `lockDuration` is enabled in `$.lockDurations[lockDuration]`.

This behavior violates the contract's time-based invariant, which mandates that lock durations must correspond to enabled periods. Consequently, users can lock tokens for arbitrary durations, including potentially maliciously long or short intervals that were not intended to be supported:

```
function releaseAndLock(
    address user,
    uint256 positionIndex,
    uint256 amount,
    uint256 lockDuration
) external nonReentrant whenNotPaused {
    // ... validation checks ...
    _release($, user, positionIndex);
    _lock($, msg.sender, user, amount, lockDuration); // Missing duration validation
}
```

This flaw allows bypassing core business logic, enabling unauthorized lock durations that could allow users to manipulate the system or lock tokens for periods inconsistent with reward calculations.

Proof of Concept

Add the following POC test function to `UsualLockup.t.sol`:

```
    function test_LockDurationValidationBypass_poc() public {
        uint256 amount = 1e18;
        uint256 unauthorizedDuration = 999 days; // This duration is NOT enabled by default

        // Verify that the unauthorized duration is not enabled
        assertEq(usualXLockup.getLockDurationStatus(unauthorizedDuration), false);

        // Create initial lock position with valid duration
        _createAliceLockPosition(amount, ONE_MONTH);

        // Wait for position to mature so it can be released
        vm.warp(block.timestamp + ONE_MONTH);

        // Prepare tokens for the new lock
        deal(address(usualX), alice, amount);
        vm.startPrank(alice);
        usualX.approve(address(usualXLockup), amount);

        // Show that direct lock() fails with unauthorized duration (expected behavior)
        vm.expectRevert(abi.encodeWithSelector(LockDurationIsNotEnabled.selector));
        usualXLockup.lock(alice, amount, unauthorizedDuration);

        // But releaseAndLock() bypasses this validation! (vulnerability)
        usualXLockup.releaseAndLock(alice, 0, amount, unauthorizedDuration);

        vm.stopPrank();

        // Verify the exploit worked - new position has unauthorized duration
```

```
        IUsualXLockup.LockPosition memory newPosition = usualXLockup.getPosition(alice, 1);
        assertEq(newPosition.endTime, block.timestamp + unauthorizedDuration);
        assertEq(newPosition.isActive, true);

        // User successfully locked tokens for 999 days, bypassing business logic!
    }
```

Output: The test passes indicating that the PoC is successful and the vulnerability exists:

```
Ran 1 test for test/vaults/UsualXLockup.t.sol:UsualXLockupUnitTest
[PASS] test_LockDurationValidationBypass_poc() (gas: 600127)
Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 5.71ms (465.38µs CPU time)
```

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:H/D:N/Y:M (8.8)

Recommendation

Insert lock duration validation before invoking `_lock()` as follows:

```
UsualXStorageV0 storage $ = _usualXStorageV0();
if ($.lockDurations[lockDuration] == false) {
    revert LockDurationIsNotEnabled();
}
_release($, user, positionIndex);
_lock($, msg.sender, user, amount, lockDuration);
```

Remediation Comment

SOLVED: The **Usual team** solved the finding by following the recommended mitigation.

Remediation Hash

https://github.com/usual-dao/core-protocol/commit/8bb2dc00b82d407459fe8d27ef62cb7daae7c26c

7.2 NO MINIMUM LOCK AMOUNT VALIDATION

// INFORMATIONAL

Description

In `UsualXLockup.sol`, The `_lock()` and public `lock()` functions currently only check that the amount is not zero but do not enforce a minimum lock amount. This allows users to lock extremely small amounts (e.g., 1 wei), which results in several issues: positions that cost more in gas to manage than their actual value, increased vulnerability to position spam attacks, inefficient reward distribution calculations, and a poor user experience due to economically unviable positions:

```
function lock(address receiver, uint256 amount, uint256 lockDuration) external {
    if (amount == 0) {
        revert AmountIsZero();
    }
    // No minimum amount validation
    _lock($, msg.sender, receiver, amount, lockDuration);
}

function _lock(
    UsualXStorageV0 storage $,
    address sender,
    address receiver,
    uint256 amount,
    uint256 lockDuration
) internal {
    // Accepts any non-zero amount
    $.usualX.safeTransferFrom(sender, address(this), amount);
    // ...
}
```

BVSS

AO:A/AC:M/AX:L/R:N/S:U/C:N/A:N/I:L/D:N/Y:N (1.7)

Recommendation

Implement validation to enforce a minimum lock amount as follows:

```
uint256 public constant MINIMUM_LOCK_AMOUNT = 1e18; // Minimum lock amount of 1 token

function lock(address receiver, uint256 amount, uint256 lockDuration) external {
    if (amount == 0) {
        revert AmountIsZero();
    }
    if (amount < MINIMUM_LOCK_AMOUNT) {
        revert AmountBelowMinimum();
    }
    // ...
}
```

Remediation Comment

ACKNOWLEDGED: The **Usual team** has acknowledged this finding.

8. AUTOMATED TESTING

Halborn utilized automated testing techniques to improve coverage of specific areas within the smart contracts under review. One of the tools employed was `Slither`, a static analysis framework for Solidity. After verifying the smart contracts in the repository and successfully compiling them into their `ABI` and binary formats, `Slither` was executed against the contracts. This tool performs static verification of mathematical relationships between Solidity variables to detect invalid or inconsistent usage of the contracts' APIs throughout the entire codebase.

The security team conducted a comprehensive review of the findings generated by the `Slither` static analysis tool. No significant issues were identified, as the reported findings were determined to be false positives.

```
UsualXLockup._topUp(UsualXLockup.UsualXLockupStorageV0,address,address,uint256,uint256) (src/vaults/UsualXLockup.sol#152-185) uses timestamp for comparisons
        Dangerous comparisons:
        - currentDay > startDay (src/vaults/UsualXLockup.sol#170)
UsualXLockup._release(UsualXLockup.UsualXLockupStorageV0,address,uint256) (src/vaults/UsualXLockup.sol#191-210) uses timestamp for comparisons
        Dangerous comparisons:
        - block.timestamp < lockPosition.endTime (src/vaults/UsualXLockup.sol#201)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#block-timestamp
INFO:Detectors:
Initializable._getInitializableStorage() (lib/openzeppelin-contracts-upgradeable/contracts/proxy/utils/Initializable.sol#223-227) uses assembly
        - INLINE ASM (lib/openzeppelin-contracts-upgradeable/contracts/proxy/utils/Initializable.sol#224-226)
PausableUpgradeable._getPausableStorage() (lib/openzeppelin-contracts-upgradeable/contracts/utils/PausableUpgradeable.sol#27-31) uses assembly
        - INLINE ASM (lib/openzeppelin-contracts-upgradeable/contracts/utils/PausableUpgradeable.sol#28-30)
ReentrancyGuardUpgradeable._getReentrancyGuardStorage() (lib/openzeppelin-contracts-upgradeable/contracts/utils/ReentrancyGuardUpgradeable.sol#46-50) uses assembly
        - INLINE ASM (lib/openzeppelin-contracts-upgradeable/contracts/utils/ReentrancyGuardUpgradeable.sol#47-49)
Address._revert(bytes) (lib/openzeppelin-contracts/contracts/utils/Address.sol#146-158) uses assembly
        - INLINE ASM (lib/openzeppelin-contracts/contracts/utils/Address.sol#151-154)
UsualXLockup._usualXLockupStorageV0() (src/vaults/UsualXLockup.sol#76-82) uses assembly
        - INLINE ASM (src/vaults/UsualXLockup.sol#79-81)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#assembly-usage
INFO:Detectors:
UsualXLockup.lock(address,uint256,uint256) (src/vaults/UsualXLockup.sol#290-307) compares to a boolean constant:
        -$.lockDurations[lockDuration] == false (src/vaults/UsualXLockup.sol#302)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#boolean-equality
```

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.