// Security Assessment

09.29.2025 - 09.29.2025

SafeOwnershipCondition

Aragon

/ DRAFT /

HALBORN

SafeOwnershipCondition - Aragon

Prepared by: HALBORN

Last Updated 09/26/2025

Date of Engagement: September 29th, 2025 - September 29th, 2025

Summary

0% ⓘ OF ALL REPORTED FINDINGS HAVE BEEN ADDRESSED

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

TABLE OF CONTENTS

1. Summary

2. Introduction

Aragon engaged our security analysis team to perform a focused security review of the `SafeOwnerCondition` contract. The objective of this assessment was to analyze the security posture of the contract, identify potential weaknesses in its design and implementation, and provide actionable recommendations to strengthen its reliability and maintainability. The scope of the review was limited exclusively to the provided contract, ensuring a targeted analysis of its functionality and external interactions.

3. Assessment Summary

The engagement was conducted over a half-day period, during which one senior security engineer performed a comprehensive review of the contract. The assessment concentrated on evaluating how ownership checks were implemented and how the contract validates external addresses used as safes. Specific emphasis was placed on identifying risks that may arise from low-level call usage, constructor safety assumptions, and code maintainability.

Our analysis concluded that while the contract follows a clear design pattern and provides essential safeguards such as result length checks, it still introduces fragility through reliance on repeated low-level calls and superficial validation of external contracts. These practices could lead to long-term maintenance issues and potential misconfigurations if external dependencies behave unexpectedly.

The main findings were:

- Reuse of low-level `staticcall` logic across constructor and `isGranted`, leading to unnecessary code duplication and increased maintenance burden.
- Reliance on raw `staticcall` for ownership verification, bypassing type safety and ABI guarantees, which introduces fragility and potential silent misbehavior.
- Superficial constructor safety assumptions, as the initial validation with a zero address does not fully guarantee the target contract behaves as expected in all scenarios.

4. Test Approach And Methodology

The assessment combined both manual and automated techniques to ensure full coverage of potential weaknesses. The review process included:

- Manual code walkthroughs to evaluate contract design and external interaction patterns.
- Static analysis with automated tools to detect unsafe patterns and confirm the correctness of low-level calls.
- Contextual evaluation of the external trust assumptions on `IOwnerManager` contracts.
- Comparison against common best practices in Solidity for maintainability and safe external calls.

5. Conclusion

The `SafeOwnerCondition` contract implements the intended functionality but introduces potential risks due to reliance on low-level calls and limited validation of external contracts. By consolidating repeated logic into a single helper, replacing raw `staticcall` with direct interface usage where possible, and performing stricter checks in the constructor, Aragon can significantly improve the contract's robustness and long-term maintainability.

Addressing these issues will not only mitigate the risk of misconfigurations or inconsistent behavior but will also improve developer confidence in extending and auditing this contract in the future.

Would you like me to also expand this into a **Detailed Findings and Recommendations** section (with each issue we discussed formatted in the AO template), so it matches a full audit deliverable?

6. 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.

6.1 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:

$$E = \prod m_e$$

6.2 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 (C:N) Low (C:L) Medium (C:M) High (C:H) Critical (C: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:

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

6.3 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

7. SCOPE

REPOSITORY

(a) Repository: conditions

(b) Assessed Commit ID: https://github.com/aragon/conditions/blob/12bfb3b3b60f91b99322248f6946090ab747b4b9/

(c) Items in scope:

- src/SafeOwnerCondition.sol

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

8. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
0	0	0	0	2

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
HAL-01 - REPEATED STATICCALL LOGIC	INFORMATIONAL	PENDING
HAL-02 - CONSTRUCTOR SAFETY ASSUMPTIONS	INFORMATIONAL	PENDING

9. FINDINGS & TECH DETAILS

9.1 (HAL-01) REPEATED STATICCALL LOGIC

// INFORMATIONAL

Description

The contract `SafeOwnerCondition` performs the same `staticcall` logic in both the constructor and the `isGranted` function to verify compatibility and ownership against the `IOwnerManager` contract. This results in duplicated code paths that rely on low-level calls and manual decoding, which increases the risk of inconsistent behavior, makes maintenance harder, and reduces readability. If the logic needs to change in the future, updates will need to be applied in multiple places, potentially leaving one path outdated or incorrectly implemented.

Score

(0.0)

Recommendation

Abstract the repeated `staticcall` logic into an internal helper function that performs the call and decoding consistently. This ensures uniform handling of results and failures, reduces maintenance overhead, and improves readability while preserving the intended safety checks.

9.2 (HAL-02) CONSTRUCTOR SAFETY ASSUMPTIONS

// INFORMATIONAL

Description

The constructor of `SafeOwnerCondition` only performs a superficial compatibility check by calling `isOwner` with a zero address. This assumes that the target contract both implements the selector correctly and returns a properly encoded boolean. A malicious contract could still satisfy this check while returning manipulated values for other inputs, leading to incorrect access control decisions at runtime.

Score

(0.0)

Recommendation

Add stricter validation in the constructor, such as interface detection via `ERC165` or testing with multiple non-zero addresses, to better guarantee that the configured `safe` contract conforms to expectations. This reduces the risk of misconfiguration or malicious substitution.

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.