// Smart Contract Security Assessment 07.22.2024 - 07.26.2024

Polygon Plugin
Aragon

HALBORN

Polygon Plugin - Aragon

Prepared by: HALBORN

Last Updated 01/30/2025

Date of Engagement by: July 22nd, 2024 - July 26th, 2024

Summary

100% ⓘ OF ALL REPORTED FINDINGS HAVE BEEN ADDRESSED

ALL FINDINGS	CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
7	0	0	3	2	2

TABLE OF CONTENTS

1. Introduction

Aragon engaged Halborn to conduct a security assessment on their smart contracts revisions beginning on July 22nd, 2024 and ending on July 26th, 2024. The security assessment was scoped to the smart contracts provided to the Halborn team.

2. Assessment Summary

The team at Halborn was provided 5 days for the engagement and assigned a full-time security engineer to evaluate 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 partially addressed by the `Aragon team`.

3. Test Approach And Methodology

Halborn performed a combination of manual and automated security testing to balance efficiency, timeliness, practicality, and accuracy regarding the scope of this assessment. While manual testing is recommended to uncover flaws in logic, process, and implementation; automated testing techniques help enhance code coverage and 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.
- Scanning of solidity files for vulnerabilities, security hot-spots or bugs. (`MythX`)
- Static Analysis of security for scoped contract, and imported functions. (`Slither`)
- Testnet deployment. (`Brownie`, `Anvil`, `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: polygon-contracts

(b) Assessed Commit ID: 75068b6

(c) Items in scope:

- PolygonMultisig.sol
- PolygonMultisigSetup.sol

Out-of-Scope: AddressList.sol, Auth.sol, DAO.sol, DaoAuthorizeUpgradeable.sol, IDAO.sol, IMemberShip.sol, IPluginSetup.sol, Multisig.sol, PermissionLib.sol, PermissionManager.sol, PluginSetup.sol, PluginUUPSUpgradeable.sol, Proposal.sol, ProposalUpgradeable.sol, Third party dependencies and economic attacks.

FILES AND REPOSITORY

(a) Repository: polygon-multithreshold-multisig

(b) Assessed Commit ID: 10d1300

(c) Items in scope:

- PolygonMultisig.sol
- PolygonMultisigSetup.sol

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

REMEDIATION COMMIT ID:

- cc25e00
- 9b81bd5
- 496bba0
- 51e2afd

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

6. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
0	0	3	2	2

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
ATTACKER COULD ABUSE VICTIM VOTE TO PASS THEIR OWN PROPOSAL	MEDIUM	SOLVED - 08/09/2024
EMERGENCY PROPOSAL DOS	MEDIUM	SOLVED - 08/09/2024
PROPOSALS LACK RESTRICTIONS FOR DURATION, PURPOSE, AND EXECUTION FEASIBILITY	MEDIUM	RISK ACCEPTED - 08/09/2024
INCONSISTENT MEMBERSHIP VERIFICATION IN PROPOSAL METADATA AND EXECUTION FUNCTIONS	LOW	SOLVED - 08/09/2024
CENTRALIZATION RISK	LOW	RISK ACCEPTED
UNINITIALIZED OPENZEPPELIN UPGRADEABLE CONTRACTS	INFORMATIONAL	ACKNOWLEDGED
USE OF MAGIC NUMBERS IN MULTISIG SETTINGS UPDATE FUNCTION	INFORMATIONAL	SOLVED - 08/09/2024

7. FINDINGS & TECH DETAILS

7.1 ATTACKER COULD ABUSE VICTIM VOTE TO PASS THEIR OWN PROPOSAL

// MEDIUM

Description

In the `PolygonMultisig` contract, proposals are created using the `createProposal()` function:

```
function createProposal(
    bytes calldata _metadata,
    IDAO.Action[] calldata _actions,
    uint256 _allowFailureMap,
    bool _approveProposal,
    uint64 _startDate,
    uint64 _endDate,
    bool _emergency
) external returns (uint256 proposalId) {
// ... (other checks and logic)
    proposalId = _createProposal({
        _creator: _msgSender(),
        _metadata: _metadata,
        _startDate: _startDate,
        _endDate: _endDate,
        _actions: _actions,
        _allowFailureMap: _allowFailureMap
    });
// ... (rest of the function)
}

##ProposalUpgradeable.sol :
import {CountersUpgradeable} from "@openzeppelin/contracts-
upgradeable/utils/CountersUpgradeable.sol";

function _createProposalId() internal returns (uint256 proposalId) {
        proposalId = proposalCount();
        proposalCounter.increment();
    }
```

Approvals are then cast using the `approve()` function:

```
function approve(uint256 _proposalId) public {
    address approver = _msgSender();
```

```
    Proposal storage proposal_ = proposals[_proposalId];
    // ... //
    proposal_.approvals += 1;
}
```

The `proposalId` is generated internally and is based on an incremental counter using OpenZeppelin's `CountersUpgradeable`. This design could potentially lead to vote misattribution in the event of a blockchain reorganization, particularly on the Polygon network where such events have occurred in the past.

Proof of Concept

In the event of a blockchain reorganization, the following scenario could occur:
1. User A submits proposal P1.
2. User B is interested in the proposal and confirms it.
3. Attacker submits proposal P2.
4. A blockchain re-org occurs. Submission of P1 is dropped in place of P2.
5. User B's confirmation is applied on top of the re-orged blockchain. Attacker gets their vote.
This vulnerability could lead to unintended approvals of malicious proposals, potentially compromising the integrity of the DAO's decision-making process.

BVSS

AO:A/AC:M/AX:M/R:N/S:C/C:N/A:N/I:C/D:N/Y:N (5.6)

Recommendation

To mitigate this risk, it is recommended to generate the `proposalId` as a hash of the proposal's properties:

- Modify the `_createProposal()` function to generate a deterministic `proposalId`:

```
function _createProposal(
    bytes calldata _metadata,
    // .. //
) internal virtual returns (uint256 proposalId) {
    proposalId = uint256(keccak256(abi.encode(
        _creator,
        _metadata,
        _startDate,
        _endDate,
        _actions,
        _allowFailureMap,
        block.number// Include block number for uniqueness
    )));
    // ... (rest of the function)
}
```

By implementing this change, approvals will be tied to the specific content of a proposal rather than a potentially mutable sequential ID, ensuring that votes cannot be misdirected even in the event of a blockchain reorganization.

Remediation

SOLVED: The **Aragon team** changed how `proposalId` indexes are computed, it is now a hash and much more secure against re-org.

Remediation Hash

https://github.com/aragon/polygon-contracts/commit/cc25e00bcbf3bb2d6d22313604d99aedc65106e9#diff-6f22234eb4cf7235c01518e005bd2491264babba6ea70feb17e5787803d79650

References

aragon/polygon-contracts/src/PolygonMultisig.sol#L275

7.2 EMERGENCY PROPOSAL DOS

// MEDIUM

Description

The `removeAddresses()` function in the `PolygonMultisig` contract lacks a crucial check to ensure that the remaining members can still meet the `emergencyMinApprovals` threshold after removal. This oversight allows for a scenario where the number of members can be reduced below the required threshold for emergency proposals, effectively causing a Denial of Service (DoS) for emergency actions.

```
function removeAddresses(
    address[] calldata _members
) external auth(UPDATE_MULTISIG_SETTINGS_PERMISSION_ID) {
    uint16 newAddresslistLength = uint16(addresslistLength() - _members.length);

// Check if the new address list length would become less than the current minimum
number of approvals required.
    if (newAddresslistLength < multisigSettings.minApprovals) {
        revert MinApprovalsOutOfBounds({
            limit: newAddresslistLength,
            actual: multisigSettings.minApprovals
        });
    }

    _removeAddresses(_members);

    emit MembersRemoved({members: _members});
}
```

The function only checks against, but`multisigSettings.minApprovals` not `multisigSettings.emergencyMinApprovals`. This allows the removal of members to a point where `newAddresslistLength < multisigSettings.emergencyMinApprovals`, creating a situation where emergency proposals cannot be executed due to insufficient members.

In scenarios where quick action is required through emergency proposals, the DAO is unable to execute these crucial actions, potentially resulting in significant financial losses or inability to respond to time-sensitive issues.

Proof of Concept

this test can be added to polygonMultisig.t.sol :

```
// forge test --match-test "test_DOS_minEmergencyApprovals"
    function test_DOS_minEmergencyApprovals() public {
        dao.grant(
```

```
            address(plugin),
            address(this),
            plugin.UPDATE_MULTISIG_SETTINGS_PERMISSION_ID()
        );

        address[] memory addrs1 = new address[](4);
        addrs1[0] = address(0xA11C3);
        addrs1[1] = address(0xA11C4);
        addrs1[2] = address(0xA11C5);
        addrs1[3] = address(0xA11C6);
        plugin.addAddresses(addrs1);

        PolygonMultisig.MultisigSettings memory multisigSettings =
PolygonMultisig.MultisigSettings({
            onlyListed: true,
            minApprovals: 1,
            emergencyMinApprovals: 3,
            delayDuration: 1 days,
            memberOnlyProposalExecution: false
        });

        console.log("Before addresslistLength = %s",plugin.addresslistLength());
        plugin.updateMultisigSettings(multisigSettings);

        address[] memory addrs2 = new address[](5);
        addrs2[0] = address(0xA11C3);
        addrs2[1] = address(0xA11C4);
        addrs2[2] = address(0xA11C5);
        addrs2[3] = address(0xA11C6);
        addrs2[4] = address(0xB0b);
        plugin.removeAddresses(addrs2);
        console.log("After  addresslistLength = %s",plugin.addresslistLength());

        vm.startPrank(address(0xdeaf));
        vm.roll(block.number + 1);
        IDAO.Action[] memory _actions = new IDAO.Action[](1);
        IDAO.Action memory _action = IDAO.Action({to: address(this), value: 0, data:
bytes("0x00")});
        _actions[0] = _action;
        plugin.createProposal({
            _metadata: bytes("ipfs://hello"),
            _actions: _actions,
            _allowFailureMap: 0,
            _approveProposal: true,
            _startDate: uint64(block.timestamp),
            _endDate: uint64(block.timestamp + 3 days),
```

```
            _emergency: true
        });
        console.log("Proposal Created but cannot be executed");
        vm.stopPrank();

        vm.expectRevert();
        plugin.execute(0);
    }
```

Result => Emergency Proposals are DOSed :

```
Ran 1 test for test/PolygonMultisig.t.sol:HalbornTest
[PASS] test_DOS_minEmergencyApprovals() (gas: 592863)
Logs:
  Before addresslistLength = 6
  After  addresslistLength = 1
  Proposal Created but cannot be executed

Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 12.63ms (2.70ms CPU time)

Ran 1 test suite in 210.80ms (12.63ms CPU time): 1 tests passed, 0 failed, 0 skipped (1 total tests)
```

BVSS

AO:S/AC:L/AX:L/C:C/I:C/A:C/D:C/Y:C/R:N/S:C (5.0)

Recommendation

It is recommended to modify the removeAddresses() function to include a check against `emergencyMinApprovals`:

```
function removeAddresses(
    address[] calldata _members
) external auth(UPDATE_MULTISIG_SETTINGS_PERMISSION_ID) {
    uint16 newAddresslistLength = uint16(addresslistLength() - _members.length);

// Check if the new address list length would become less than the current minimum
number of approvals required.
    if (newAddresslistLength < multisigSettings.minApprovals || newAddresslistLength
< multisigSettings.emergencyMinApprovals) {
        revert MinApprovalsOutOfBounds({
            limit: newAddresslistLength,
            actual: (multisigSettings.minApprovals >
multisigSettings.emergencyMinApprovals) ? multisigSettings.minApprovals :
multisigSettings.emergencyMinApprovals
        });
    }

    _removeAddresses(_members);
```

```
    emit MembersRemoved({members: _members});
}
```

Remediation

SOLVED : The **Aragon team** added a check to prevent removing addresses below the emergency approvals' threshold.

Remediation Hash

https://github.com/aragon/polygon-contracts/commit/9b81bd511d453623082b3048f16b78430d233540

References

aragon/polygon-contracts/src/PolygonMultisig.sol#L240

7.3 PROPOSALS LACK RESTRICTIONS FOR DURATION, PURPOSE, AND EXECUTION FEASIBILITY

// MEDIUM

Description

The `PolygonMultisig` contract exhibits four issues related to proposal management:

1. Indefinite Proposal Lifespan: The `createProposal` function allows for arbitrary end times without enforcing a maximum duration.
2. Unbounded Number of Actions: The contract allows an unlimited number of actions per proposal.
3. Actionless Proposals: The contract permits the creation of proposals without any defined actions.
4. Infeasible Non-Emergency Proposals: The contract allows the creation of non-emergency proposals with an end date less than the `delayDuration` away from the start date, making them impossible to execute due to a mandatory delay of 1 day.

```
function createProposal(
        bytes calldata _metadata,
        IDAO.Action[] calldata _actions,
        uint256 _allowFailureMap,
        bool _approveProposal,
        uint64 _startDate,
        uint64 _endDate,
        bool _emergency
    ) external returns (uint256 proposalId) {
        // .. //

        // @audit no limit on endDate
        if (_endDate < _startDate) {
            revert DateOutOfBounds({limit: _startDate, actual: _endDate});
        }

        // .. //

        // Create the proposal
        Proposal storage proposal_ = proposals[proposalId];
        proposal_.metadata = _metadata;

        proposal_.parameters.snapshotBlock = snapshotBlock;
        proposal_.parameters.startDate = _startDate;
        proposal_.parameters.endDate = _endDate;
        proposal_.parameters.minApprovals = multisigSettings.minApprovals;
        // setting the new data
        proposal_.parameters.emergency = _emergency;
```

```
        proposal_.parameters.emergencyMinApprovals =
multisigSettings.emergencyMinApprovals;
        proposal_.parameters.delayDuration = multisigSettings.delayDuration;
        proposal_.parameters.memberOnlyProposalExecution = multisigSettings
            .memberOnlyProposalExecution;

        // .. //

        for (uint256 i; i < _actions.length; ) {
            proposal_.actions.push(_actions[i]);
            unchecked {
                ++i;
            }
        }

        // .. //
    }
```

Impacts :

- Non-emergency proposals can be created with a duration shorter than the `delayDuration`, rendering them impossible to execute. This wastes resources and can lead to confusion among DAO members.
- Increased Attack Surface: More complex proposals provide more opportunities for subtle manipulations or exploits.
- The DAO's state can be manipulated through the execution of proposals that no longer align with the current context or members' intentions.
- Review Complexity: Approvers and confirmers must review a potentially large number of actions, increasing the likelihood of oversight or misunderstanding.

Proof of Concept

This test can be added to polygonMultisig.t.sol => It creates an action for a proposal with very very very long endDate,a proposal with 12_000 actions, a proposal with less endDate than delay, and a proposal with no actions in it :

```
// forge test --match-test "test_action_Missing_Restrictions"
    function test_action_Missing_Restrictions() public {
        vm.deal(address(0xB0b),1 ether);
        vm.deal(address(dao),1 ether);
        vm.startPrank(address(0xB0b));

        // 1. INFEASIBLE PROPOSAL DUE TO DELAY
        IDAO.Action[] memory _actions1 = new IDAO.Action[](1);
        IDAO.Action memory _action = IDAO.Action({to: address(this), value: 0, data:
bytes("0x00")});
        _actions1[0] = _action;
```

```
        plugin.createProposal({
            _metadata: bytes("ipfs://hello"),
            _actions: _actions1,
            _allowFailureMap: 0,
            _approveProposal: true,
            _startDate: uint64(block.timestamp),
            _endDate: uint64(block.timestamp + (1 days) / 2),
            _emergency: false
        });
        plugin.startProposalDelay(0,bytes("0x02"));
        vm.expectRevert();
        plugin.confirm(0);
        vm.warp(block.timestamp + 1 days);
        vm.expectRevert();
        plugin.confirm(0);
        vm.expectRevert();
        plugin.execute(0);


        // 2. LOT OF ACTIONS
        IDAO.Action[] memory _actionsBig = new IDAO.Action[](12_000);
        IDAO.Action memory _action2 = IDAO.Action({to: address(this), value: 0, data:
bytes("0x00")});
        _actionsBig[0] = _action2;
        plugin.createProposal({
            _metadata: bytes("ipfs://hello"),
            _actions: _actionsBig,
            _allowFailureMap: 0,
            _approveProposal: true,
            _startDate: uint64(block.timestamp),
            _endDate: uint64(type(uint256).max),
            _emergency: true
        });


        // 3. EMPTY ACTIONS
        IDAO.Action[] memory _actionsEmpty = new IDAO.Action[](0);
        plugin.createProposal({
            _metadata: bytes("ipfs://hello"),
            _actions: _actionsEmpty,
            _allowFailureMap: 0,
            _approveProposal: true,
            _startDate: uint64(block.timestamp),
            _endDate: uint64(type(uint256).max),
            _emergency: true
        });
```

```
        vm.stopPrank();

    }
```

It can be observed that proposals are not restricted for certain requirements :

```
➜  polygon-contracts git:(main) ✗ forge test --match-test "test_action_Missing_Restrictions"
[⠒] Compiling...
No files changed, compilation skipped

Ran 1 test for test/PolygonMultisig.t.sol:HalbornTest
[PASS] test_action_Missing_Restrictions() (gas: 158746436)
Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 127.13ms (118.28ms CPU time)

Ran 1 test suite in 206.85ms (127.13ms CPU time): 1 tests passed, 0 failed, 0 skipped (1 total tests)
➜  polygon-contracts git:(main) ✗
```

BVSS

AO:A/AC:L/AX:L/C:L/I:L/A:L/D:N/Y:N/R:N/S:C (4.7)

Recommendation

To address this vulnerability, it is recommended to implement the following changes:
1. Introduce a maximum proposal duration in the `MultisigSettings` struct and enforce it in the `createProposal` function.
2. Add a check in the `createProposal` function to ensure non-emergency proposals have a duration of at least delayDuration + 1 (with a new constant variable for min duration).
3. Even though a maximum duration is enforced in DAO.sol it would be better to have it in PolygonMultisig.sol also.
4. Implement a check to ensure that every proposal contains at least one action.

Remediation

RISK ACCEPTED: The **Aragon team** accepted the risks 1,3,4 but added a minimal extra duration to remediate risk.

Remediation Hash

https://github.com/aragon/polygon-contracts/commit/496bba0c652055ba7487863fcbe029b476ca0193

References

aragon/polygon-contracts/src/PolygonMultisig.sol#L275C5-L277C14

7.4 INCONSISTENT MEMBERSHIP VERIFICATION IN PROPOSAL METADATA AND EXECUTION FUNCTIONS

// LOW

Description

The `PolygonMultisig` contract inconsistently applies membership checks in critical functions. Specifically, the `_checkProposalForMetadata` and `execute` functions use `isListed()` to verify membership, which checks the current state rather than using `isListedAtBlock` which is used to check if a member can approve or confirm a proposal.

`_checkProposalForMetadata` and `execute` functions should use `isListedAtBlock()` to verify membership at the proposal's snapshot block, ensuring consistency with the proposal's creation time, approvals and confirmations.

The problematic code in `_checkProposalForMetadata`:

```
function _checkProposalForMetadata(
    uint256 _proposalId
) internal view returns (Proposal storage) {
    if (!isListed(_msgSender())) {
        revert NotInMemberList(_msgSender());
    }
// ... rest of the function
}
```

Similarly in `execute()` function :

```
function execute(uint256 _proposalId) public {
    if (!_canExecute(_proposalId)) {
        revert ProposalExecutionForbidden(_proposalId);
    }

    _execute(_proposalId);
}

function _canExecute(uint256 _proposalId) internal view returns (bool) {
    Proposal storage proposal_ = proposals[_proposalId];

// ... other checks

    if (proposal_.parameters.memberOnlyProposalExecution && !isListed(_msgSender()))
{
        return false;
```

```
    }

// ... rest of the function
}
```

BVSS

AO:A/AC:L/AX:L/C:N/I:L/A:N/D:N/Y:N/R:N/S:C (3.1)

Recommendation

It is recommended to modify the `checkProposalForMetadata` *and* `execute` *(via* `canExecute`) functions to use `isListedAtBlock()` instead of `isListed()`. This ensures that membership is checked against the proposal's snapshot block, maintaining temporal consistency.

Remediation

SOLVED : `isListedAtBlock` is now used instead of `isListed()`.

Remediation Hash

https://github.com/aragon/polygon-contracts/commit/51e2afddd6de74170cdc4374cbfb778c0348727e

References

aragon/polygon-contracts/src/PolygonMultisig.sol#L454
aragon/polygon-contracts/src/PolygonMultisig.sol#L611

7.5 CENTRALIZATION RISK

// LOW

Description

The `PolygonMultisig` contract grants extensive control to the holder of the `UPDATE_MULTISIG_SETTINGS_PERMISSION_ID` role. This role has the ability to modify critical multisig parameters, add and remove addresses of the list without any checks. The following functions are under the control of this role:

```
function addAddresses(
    address[] calldata _members
) external auth(UPDATE_MULTISIG_SETTINGS_PERMISSION_ID) {
// ...
}

function removeAddresses(
    address[] calldata _members
) external auth(UPDATE_MULTISIG_SETTINGS_PERMISSION_ID) {
// ...
}

function updateMultisigSettings(
    MultisigSettings calldata _multisigSettings
) external auth(UPDATE_MULTISIG_SETTINGS_PERMISSION_ID) {
// ...
}
```

Listed addresses can approve, confirm and execute proposals.

Furthermore, the contract allows for the creation, approval, and execution of emergency proposals within a single block and the deployment script sets dangerously low approval thresholds:

```
PolygonMultisig.MultisigSettings memory multisigSettings = PolygonMultisig
    .MultisigSettings({
        onlyListed: true,
        minApprovals: 1,
        emergencyMinApprovals: 1,
        delayDuration: 1 days,
        memberOnlyProposalExecution: false
    });
```

This configuration creates a severe centralization risk:

1. A single malicious actor with the `UPDATE_MULTISIG_SETTINGS_PERMISSION_ID` role can change the multisig composition and settings.
2. With `minApprovals` set to 1, a single member can add and remove any members from the approved list and create/execute emergency proposals without oversight.
3. These factors combined allow for a **single malicious** `UPDATE_MULTISIG_SETTINGS_PERMISSION_ID` role owner to do some irreversible changes to the DAO's structure and operations

Proof of Concept

This test can be added to PolygonMultisig.t.sol :

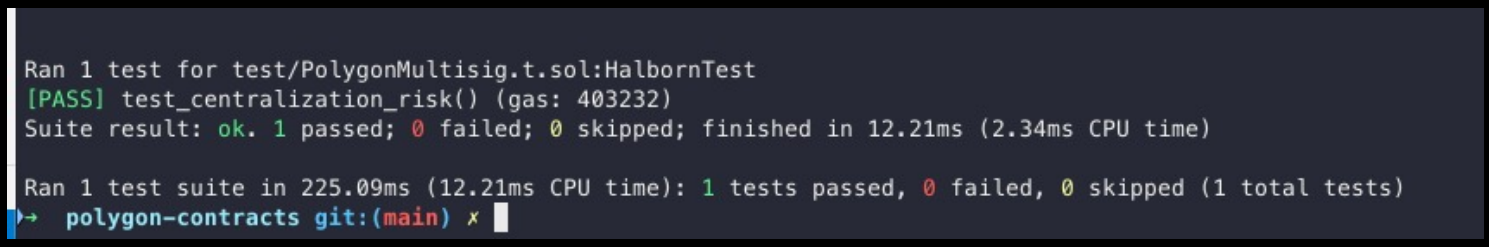
```
// forge test --match-test "test_centralization_risk"
    function test_centralization_risk() public {
        dao.grant(
            address(plugin),
            address(this),
            plugin.UPDATE_MULTISIG_SETTINGS_PERMISSION_ID()
        );
        console.log("List modification timestamp     : %s",block.timestamp);
        address[] memory addrs1 = new address[](1);
        addrs1[0] = address(this);
        plugin.addAddresses(addrs1);

        address[] memory addrs2 = new address[](2);
        // Bob removes all other users except him
        addrs2[0] = address(0xdeaf);
        addrs2[1] = address(0xB0b);
        plugin.removeAddresses(addrs2);

        // Then Bob creates a malicious proposal and execute it
        IDAO.Action[] memory _actions = new IDAO.Action[](0);
        vm.roll(block.number+1);
        console.log("Timestamp of creation proposal  : %s",block.timestamp);
        plugin.createProposal({
            _metadata: bytes("ipfs://hello"),
            _actions: _actions,
            _allowFailureMap: 0,
            _approveProposal: true,
            _startDate: uint64(block.timestamp),
            _endDate: uint64(block.timestamp + 1 days),
            _emergency: true
        });
        (
            bool _executed,
            uint16 _approvals,
            PolygonMultisig.ProposalParameters memory _parameters,
            IDAO.Action[] memory _actions2,
```

```
            uint256 _allowFailureMap,
            uint16 _confirmations,
            bytes memory _metadata,
            bytes memory _secondaryMetadata,
            uint64 _firstDelayStartBlock
        ) = plugin.getProposal(0);
        console.log("Timestamp of creation execution : %s",block.timestamp);
        //console.log("Start Date : %s",_parameters.startDate);
        //console.log("snapshotBlock : %s",_parameters.snapshotBlock);
        //console.log("IsListed : %s",plugin.isListed(address(this)));
        //console.log("IsListedAtBlock :
%s",plugin.isListedAtBlock(address(this),_parameters.snapshotBlock));
        plugin.execute(0);
    }
```

As it can be seen in the screenshot the test pass and the multisig could be destroyed by the proposal executed



BVSS

AO:S/AC:L/AX:L/C:N/I:H/A:H/D:N/Y:N/R:N/S:C (2.3)

Recommendation

To mitigate these risks, it is recommended to implement the following changes:

- Introduce a time-lock for changes made by the `UPDATE_MULTISIG_SETTINGS_PERMISSION_ID` role (even for list addresses change).
- Create a different role for updating approved lists to separate multisig powers update from list modifications.
- Enforce a minimum number of approvals greater than 1 in the contract(deployment script).

Remediation

RISK ACCEPTED: The **Aragon team** accepted the risk but implemented a real variable for the minimum approval threshold.

References

aragon/polygon-contracts/src/PolygonMultisig.sol#L221
aragon/polygon-contracts/src/PolygonMultisig.sol#L240
aragon/polygon-contracts/src/PolygonMultisig.sol#L260

7.6 UNINITIALIZED OPENZEPPELIN UPGRADEABLE CONTRACTS

// INFORMATIONAL

Description

The `PolygonMultisig` contract inherits from multiple OpenZeppelin upgradeable contracts, including `ContextUpgradeable`, `UUPSUpgradeable`, and `ERC165Upgradeable`. However, the initialization functions for these contracts are not called during the PolygonMultisig initialization process. This oversight can be seen in the initialize function of PolygonMultisig:

```
function initialize(
    IDAO _dao,
    address[] calldata _members,
    MultisigSettings calldata _multisigSettings
) external initializer {
    __PluginUUPSUpgradeable_init(_dao);
    // Missing initializations:// __Context_init_unchained()
    // __UUPSUpgradeable_init()// __ERC165_init()// ...
}
```

The following initializations are missing:
1. ContextUpgradeable: https://github.com/OpenZeppelin/openzeppelin-contracts-upgradeable/blob/v4.7.2/contracts/utils/ContextUpgradeable.sol#L18
2. UUPSUpgradeable: https://github.com/OpenZeppelin/openzeppelin-contracts-upgradeable/blob/v4.7.2/contracts/proxy/utils/UUPSUpgradeable.sol#L23
3. ERC165Upgradeable: https://github.com/OpenZeppelin/openzeppelin-contracts-upgradeable/blob/v4.7.2/contracts/utils/introspection/ERC165Upgradeable.sol#L24

Score

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

Recommendation

Even if no code is currently written in these OpenZeppelin initializers, It is recommended to properly initialize all inherited OpenZeppelin upgradeable contracts in case of further updates adding code to them.

Remediation

ACKNOWLEDGED: The **Aragon team** acknowledged the finding but does not plan to remediate it.

References

aragon/polygon-contracts/src/PolygonMultisig.sol#L189

7.7 USE OF MAGIC NUMBERS IN MULTISIG SETTINGS UPDATE FUNCTION

// INFORMATIONAL

Description

The `_updateMultisigSettings` function in the `PolygonMultisig` contract uses a hard-coded value of 1 as the minimum threshold for both `minApprovals` and `emergencyMinApprovals`. This practice of using magic numbers reduces code readability and maintainability:

```
    function _updateMultisigSettings(MultisigSettings calldata _multisigSettings)
internal {

        // .. //
        // @audit should enforce a real variable , not an arbitrarily "1"
        if (_multisigSettings.minApprovals < 1 ||
_multisigSettings.emergencyMinApprovals < 1) {
            revert MinApprovalsOutOfBounds({limit: 1, actual:
_multisigSettings.minApprovals});
        }

        // .. //
    }
```

The hard-coded value 1 is used to enforce a minimum threshold for minApprovals. However, this value is not explicitly defined as a constant and is definitely low for approvals that can be approved by proposers itself.

Score

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

Recommendation

It is recommended to define explicit constants for the minimum thresholds (at least >= 2) and use these constants in the function `_updateMultisigSettings`.

Remediation

SOLVED : Explicit constants are now used.

Remediation Hash

https://github.com/aragon/polygon-contracts/commit/496bba0c652055ba7487863fcbe029b476ca0193

References

aragon/polygon-contracts/src/PolygonMultisig.sol#L635

8. AUTOMATED TESTING

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.

```
↦  polygon-contracts git:(main) ✗ slither . --exclude-low
'forge clean' running (wd: /Users/liliancariou/Desktop/Halborn/audits/polygon-contracts)
'forge config --json' running
'forge build --build-info --skip */test/** */script/** --force' running (wd: /Users/liliancariou/Desktop/Halborn/audits/polygon-contracts)
INFO:Detectors:
ERC1967UpgradeUpgradeable._functionDelegateCall(address,bytes) (lib/openzeppelin-contracts-upgradeable/contracts/proxy/ERC1967/ERC1967UpgradeUpgradeable.sol#198-204) uses delegat
ecall to a input-controlled function id
        - (success,returndata) = target.delegatecall(data) (lib/openzeppelin-contracts-upgradeable/contracts/proxy/ERC1967/ERC1967UpgradeUpgradeable.sol#202)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#controlled-delegatecall
INFO:Detectors:
MathUpgradeable.mulDiv(uint256,uint256,uint256) (lib/openzeppelin-contracts-upgradeable/contracts/utils/math/MathUpgradeable.sol#55-135) has bitwise-xor operator ^ instead of the
 exponentiation operator **:
         - inverse = (3 * denominator) ^ 2 (lib/openzeppelin-contracts-upgradeable/contracts/utils/math/MathUpgradeable.sol#117)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#incorrect-exponentiation
INFO:Detectors:
MathUpgradeable.mulDiv(uint256,uint256,uint256) (lib/openzeppelin-contracts-upgradeable/contracts/utils/math/MathUpgradeable.sol#55-135) performs a multiplication on the result o
f a division:
        - denominator = denominator / twos (lib/openzeppelin-contracts-upgradeable/contracts/utils/math/MathUpgradeable.sol#102)
        - inverse = (3 * denominator) ^ 2 (lib/openzeppelin-contracts-upgradeable/contracts/utils/math/MathUpgradeable.sol#117)
MathUpgradeable.mulDiv(uint256,uint256,uint256) (lib/openzeppelin-contracts-upgradeable/contracts/utils/math/MathUpgradeable.sol#55-135) performs a multiplication on the result o
f a division:
        - denominator = denominator / twos (lib/openzeppelin-contracts-upgradeable/contracts/utils/math/MathUpgradeable.sol#102)
        - inverse *= 2 - denominator * inverse (lib/openzeppelin-contracts-upgradeable/contracts/utils/math/MathUpgradeable.sol#121)
MathUpgradeable.mulDiv(uint256,uint256,uint256) (lib/openzeppelin-contracts-upgradeable/contracts/utils/math/MathUpgradeable.sol#55-135) performs a multiplication on the result o
f a division:
        - denominator = denominator / twos (lib/openzeppelin-contracts-upgradeable/contracts/utils/math/MathUpgradeable.sol#102)
        - inverse *= 2 - denominator * inverse (lib/openzeppelin-contracts-upgradeable/contracts/utils/math/MathUpgradeable.sol#122)
MathUpgradeable.mulDiv(uint256,uint256,uint256) (lib/openzeppelin-contracts-upgradeable/contracts/utils/math/MathUpgradeable.sol#55-135) performs a multiplication on the result o
f a division:
        - denominator = denominator / twos (lib/openzeppelin-contracts-upgradeable/contracts/utils/math/MathUpgradeable.sol#102)
        - inverse *= 2 - denominator * inverse (lib/openzeppelin-contracts-upgradeable/contracts/utils/math/MathUpgradeable.sol#123)
MathUpgradeable.mulDiv(uint256,uint256,uint256) (lib/openzeppelin-contracts-upgradeable/contracts/utils/math/MathUpgradeable.sol#55-135) performs a multiplication on the result o
f a division:
        - denominator = denominator / twos (lib/openzeppelin-contracts-upgradeable/contracts/utils/math/MathUpgradeable.sol#102)
        - inverse *= 2 - denominator * inverse (lib/openzeppelin-contracts-upgradeable/contracts/utils/math/MathUpgradeable.sol#124)
MathUpgradeable.mulDiv(uint256,uint256,uint256) (lib/openzeppelin-contracts-upgradeable/contracts/utils/math/MathUpgradeable.sol#55-135) performs a multiplication on the result o
f a division:
        - denominator = denominator / twos (lib/openzeppelin-contracts-upgradeable/contracts/utils/math/MathUpgradeable.sol#102)
        - inverse *= 2 - denominator * inverse (lib/openzeppelin-contracts-upgradeable/contracts/utils/math/MathUpgradeable.sol#125)
MathUpgradeable.mulDiv(uint256,uint256,uint256) (lib/openzeppelin-contracts-upgradeable/contracts/utils/math/MathUpgradeable.sol#55-135) performs a multiplication on the result o
f a division:
        - denominator = denominator / twos (lib/openzeppelin-contracts-upgradeable/contracts/utils/math/MathUpgradeable.sol#102)
        - inverse *= 2 - denominator * inverse (lib/openzeppelin-contracts-upgradeable/contracts/utils/math/MathUpgradeable.sol#126)
MathUpgradeable.mulDiv(uint256,uint256,uint256) (lib/openzeppelin-contracts-upgradeable/contracts/utils/math/MathUpgradeable.sol#55-135) performs a multiplication on the result o
f a division:
        - prod0 = prod0 / twos (lib/openzeppelin-contracts-upgradeable/contracts/utils/math/MathUpgradeable.sol#105)
        - result = prod0 * inverse (lib/openzeppelin-contracts-upgradeable/contracts/utils/math/MathUpgradeable.sol#132)
```

```
        - inverse *= 2 - denominator * inverse (lib/openzeppelin-contracts-upgradeable/contracts/utils/math/MathUpgradeable.sol#126)
MathUpgradeable.mulDiv(uint256,uint256,uint256) (lib/openzeppelin-contracts-upgradeable/contracts/utils/math/MathUpgradeable.sol#55-135) performs a multiplication on the result o
f a division:
        - prod0 = prod0 / twos (lib/openzeppelin-contracts-upgradeable/contracts/utils/math/MathUpgradeable.sol#105)
        - result = prod0 * inverse (lib/openzeppelin-contracts-upgradeable/contracts/utils/math/MathUpgradeable.sol#132)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#divide-before-multiply
INFO:Detectors:
CheckpointsUpgradeable.push(CheckpointsUpgradeable.History,uint256) (lib/openzeppelin-contracts-upgradeable/contracts/utils/CheckpointsUpgradeable.sol#60-71) uses a dangerous str
ict equality:
        - pos > 0 && self._checkpoints[pos - 1]._blockNumber == block.number (lib/openzeppelin-contracts-upgradeable/contracts/utils/CheckpointsUpgradeable.sol#63)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#dangerous-strict-equalities
INFO:Detectors:
ERC1967Upgrade._upgradeToAndCall(address,bytes,bool) (lib/openzeppelin-contracts/contracts/proxy/ERC1967/ERC1967Upgrade.sol#65-74) ignores return value by Address.functionDelegat
eCall(newImplementation,data) (lib/openzeppelin-contracts/contracts/proxy/ERC1967/ERC1967Upgrade.sol#72)
ERC1967Upgrade._upgradeBeaconToAndCall(address,bytes,bool) (lib/openzeppelin-contracts/contracts/proxy/ERC1967/ERC1967Upgrade.sol#174-184) ignores return value by Address.functio
nDelegateCall(IBeacon(newBeacon).implementation(),data) (lib/openzeppelin-contracts/contracts/proxy/ERC1967/ERC1967Upgrade.sol#182)
Addresslist._addAddresses(address[]) (lib/osx/packages/contracts/src/plugins/utils/Addresslist.sol#60-74) ignores return value by _addresslistCheckpoints[_newAddresses[i]].push(1
) (lib/osx/packages/contracts/src/plugins/utils/Addresslist.sol#67)
Addresslist._addAddresses(address[]) (lib/osx/packages/contracts/src/plugins/utils/Addresslist.sol#60-74) ignores return value by _addresslistLengthCheckpoints.push(_uncheckedAdd
,_newAddresses.length) (lib/osx/packages/contracts/src/plugins/utils/Addresslist.sol#73)
Addresslist._removeAddresses(address[]) (lib/osx/packages/contracts/src/plugins/utils/Addresslist.sol#78-92) ignores return value by _addresslistCheckpoints[_exitingAddresses[i]]
.push(0) (lib/osx/packages/contracts/src/plugins/utils/Addresslist.sol#85)
Addresslist._removeAddresses(address[]) (lib/osx/packages/contracts/src/plugins/utils/Addresslist.sol#78-92) ignores return value by _addresslistLengthCheckpoints.push(_unchecked
Sub,_exitingAddresses.length) (lib/osx/packages/contracts/src/plugins/utils/Addresslist.sol#91)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#unused-return
```

All issues identified by `Slither` were proved to be false positives or have been added to the issue list in this report.

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.