President Elector - Findings Report

Table of contents

Contest Summary

Sponsor: First Flight #24

Dates: Sep 12th, 2024 - Sep 19th, 2024

See more contest details here

Results Summary

Number of findings:

- High: 3
- Medium: 4
- Low: 6

High Risk Findings

H-01. Reentrancy Risk in `selectPresident`

Description: The `selectPresident` function modifies critical state variables before completing all logic. If future modifications introduce external calls, this could be exploited to manipulate the election process through reentrancy.

Impact:

- *State Manipulation:* If external calls were introduced, an attacker could potentially reenter the function and alter the election results by manipulating state variables such as `s_currentPresident`, `s_candidateList`, and `s_previousVoteEndTimeStamp`.

Proof of Concept: Currently, no external calls exist in the function, so direct exploitation isn't possible. However, if external calls were added, reentrancy could be exploited as follows:

```
    contract Malicious {
        RankedChoice rankedChoice;
        bool reentered = false;

        constructor(address _rankedChoice) {
            rankedChoice = RankedChoice(_rankedChoice);
        }

        function attack() external {
            rankedChoice.selectPresident();
        }

        fallback() external payable {
            if (!reentered) {
                reentered = true;
                rankedChoice.selectPresident();
            }
        }
    }
```

Recommended Mitigation:

- *Use the "Checks-Effects-Interactions" Pattern:* Ensure all state changes occur after all logic is complete and before any external calls.
- *Reentrancy Guard:* Consider using a reentrancy guard (such as OpenZeppelin's ReentrancyGuard) to prevent reentrant calls.

H-02. Lack of Input Validation in `rankCandidates`

Description: The `rankCandidates` function allows users to submit an ordered list of candidate addresses without validating the uniqueness or validity of these addresses. This means that a voter can submit a list with duplicate addresses or addresses that are not valid candidates.

Impact:

- *Skewed Voting Results:* Voters can unfairly influence the election by ranking the same candidate multiple times, effectively giving them more weight in the election process.
- *Invalid Candidates:* Addresses that are not valid candidates can be included in the rankings, potentially disrupting the election process and results.

Proof of Concept: A voter could call the `rankCandidates` function with a list like `[candidate1, candidate1, candidate2]`, which would unfairly give more weight to `candidate1`:

```
    address[] memory orderedCandidates = new address[]();
    orderedCandidates[0] = candidate1;
    orderedCandidates[1] = candidate1; // Duplicate entry
    orderedCandidates[2] = candidate2;

    // Assuming `rankedChoice` is an instance of the RankedChoice contract
    rankedChoice.rankCandidates(orderedCandidates);
```

Recommended Mitigation:

- *Uniqueness Check:* Implement a mechanism to ensure that all addresses in orderedCandidates are unique. This can be done using a mapping to track the presence of each address.
- *Validity Check:* Ensure that all addresses in orderedCandidates are valid candidates. This could involve maintaining a list or mapping of valid candidates and checking against it.

Example Mitigation Code

```
    function _rankCandidates(
        address[] memory orderedCandidates,
        address voter
    ) internal {
        // Checks
        if (orderedCandidates.length > MAX_CANDIDATES) {
            revert RankedChoice__InvalidInput();
        }
        if (!_isInArray(VOTERS, voter)) {
            revert RankedChoice__InvalidVoter();
```

```
        }

        // Uniqueness and Validity Check
        mapping(address => bool) memory seenCandidates;
        for (uint256 i = 0; i < orderedCandidates.length; i++) {
            address candidate = orderedCandidates[i];

            // Check for uniqueness
            if (seenCandidates[candidate]) {
                revert RankedChoice__InvalidInput(); // Duplicate candidate
            }
            seenCandidates[candidate] = true;

            // Check for validity
            if (!_isInArray(s_candidateList, candidate)) {
                revert RankedChoice__InvalidInput(); // Invalid candidate
            }
        }

        // Internal Effects
        s_rankings[voter][s_voteNumber] = orderedCandidates;
    }
```

H-03. Potential Denial of Service (DoS) in `_selectPresidentRecursive`

Description: The `_selectPresidentRecursive` function uses recursion to process candidates, which could lead to hitting the gas limit if there are a large number of candidates or voters. This could result in a denial of service, preventing the function from completing successfully.

Impact:

- *Denial of Service (DoS):* If the number of candidates or voters is large, the recursive calls could consume all available gas, causing the transaction to fail. This would prevent the election process from completing, potentially halting the selection of a new president.

Proof of Concept: While a direct proof of concept in Solidity is challenging due to gas limits, the issue can be illustrated by attempting to process a large number of candidates or voters, leading to out-of-gas errors.

Recommended Mitigation:

- *Iterative Approach:* Replace the recursive logic with an iterative approach to avoid deep call stacks and excessive gas consumption.
- *Batch Processing:* If the number of candidates or voters is expected to be very large, consider processing them in smaller batches over multiple transactions.

Medium Risk Findings

M-01. Lack of Access Control for `selectPresident`

Description: The `selectPresident` function can be called by any external account, which means any user can trigger the election process to select a new president. This lack of access control could lead to unauthorized or premature elections.

Impact:

- *Unauthorized Elections:* Without proper access control, any user can trigger the election process, potentially disrupting the intended election schedule or process.
- *Premature Elections:* Users could call selectPresident before the community is ready, leading to unexpected changes in leadership.

Proof of Concept: Any user can call the function without restriction:

```
    rankedChoice.selectPresident();
```

Recommended Mitigation:

- *Access Control:* Implement access control to restrict who can call the selectPresident function. This could be limited to a specific role or set of addresses (e.g., an admin or governance contract).
- *Time-Based Restrictions:* Ensure that the function can only be called after a certain period or under specific conditions to prevent premature elections.

M-02. Potential Gas Limit Issues with Large Voter or Candidate Lists

Description: The contract processes all voters and candidates in loops within functions like `selectPresident` and `_selectPresidentRecursive`. If the number of voters or candidates is large, these loops could exceed the block gas limit, causing transactions to fail.

Impact:

- *Transaction Failure:* Large voter or candidate lists could lead to transactions running out of gas, resulting in failed attempts to execute critical functions like selecting a president.
- *Operational Disruption:* This could disrupt the election process, preventing the selection of a new president and potentially halting the governance process.

Proof of Concept: If the number of voters or candidates is sufficiently large, attempting to execute the selectPresident function could result in an out-of-gas error:

```
    // Assuming a large number of voters and candidates
    rankedChoice.selectPresident();
```

Recommended Mitigation:

- *Batch Processing:* Process voters and candidates in smaller batches over multiple transactions to avoid hitting gas limits.

- *Gas Limit Checks:* Implement checks to ensure that loops do not exceed a safe number of iterations based on current gas limits.
- *Optimized Data Structures:* Use more gas-efficient data structures and algorithms to handle large lists.

M-03. Lack of Event Emission for Critical State Changes

Description: The `RankedChoice` contract does not emit events for critical state changes, such as when a new president is selected or when candidates are ranked. Events are crucial for tracking changes and providing transparency and auditability in smart contracts.

Impact:

- *Lack of Transparency:* Without events, it is difficult for off-chain systems and users to track changes and actions within the contract.
- *Auditability Issues:* Monitoring and auditing the contract's behavior becomes challenging, as there is no record of critical state changes.
- *User Notification:* Users and external systems cannot be easily notified of important changes, such as the election of a new president.

Proof of Concept: Currently, the contract does not emit any events, so there is no way to track when a new president is selected or when candidates are ranked.

Recommended Mitigation:

- *Emit Events for Critical Actions:* Emit events for actions such as selecting a new president, ranking candidates, and any other significant state changes.
- *Define Clear Event Structures:* Define events with clear and informative parameters to capture relevant details of the state changes.

M-04. Inadequate Handling of Ties in Candidate Selection

Description: The current implementation of the `RankedChoice` contract does not explicitly handle tie situations where two or more candidates receive the same number of votes in a round. This can lead to unexpected behavior or failure to resolve the election process.

Impact:

- *Indeterminate Election Outcome:* If a tie occurs, the contract may not be able to determine a clear winner, potentially stalling the election process.
- *Unexpected Behavior:* The contract might revert or behave unpredictably if it encounters a tie, leading to a lack of resolution in the election.

Proof of Concept: If two candidates receive the same number of votes in a round, the current logic does not specify how to proceed, which can result in an unresolved state:

```
    // Assume candidate1 and candidate2 receive the same number of votes
    // The contract does not handle this tie situation explicitly
```

Recommended Mitigation:

- *Implement Tie-Breaking Logic: *Introduce a mechanism to handle ties, such as a runoff election, random selection, or predefined rules to break ties.
- *Define Clear Rules for Ties:* Clearly define and document the rules for handling ties to ensure predictable and transparent outcomes.

Low Risk Findings

L-01. Use of Private Visibility for State Variables

Description: The contract uses private visibility for several state variables, such as `s_currentPresident`, `s_previousVoteEndTimeStamp`, and `s_voteNumber`. While this restricts access to these variables from outside the contract, it also limits transparency and may hinder testing and debugging efforts.

Impact:

- *Reduced Transparency:* External users and other contracts cannot access these variables, which may be necessary for transparency and integration purposes.
- *Testing and Debugging Challenges:* Developers may find it difficult to test and debug the contract without visibility into these state variables.

Proof of Concept: The following state variables are declared as private, limiting their visibility:

```
    address private s_currentPresident;
    uint256 private s_previousVoteEndTimeStamp;
    uint256 private s_voteNumber;
```

Recommended Mitigation:

- *Consider Using internal or public Visibility:*
    - Use internal if the variables need to be accessed by derived contracts.
    - Use public if the variables should be accessible externally for transparency and integration purposes.
- *Provide Getter Functions:* If maintaining private visibility is necessary, consider providing public getter functions to allow external access to these variables.

L-02. Inefficient Use of Storage in Candidate Lists

Description: The `RankedChoice` contract uses dynamic arrays for storing candidate lists and rankings, which can lead to inefficient use of storage. Dynamic arrays in Solidity can be costly in terms of gas when resizing or iterating over them, especially if the arrays grow large.

Impact:

- *Increased Gas Costs:* Operations involving dynamic arrays, such as adding or removing elements, can become expensive, leading to higher transaction costs.

- *Potential Performance Bottlenecks:* As the number of candidates or voters increases, operations on these arrays may become slower, affecting the contract's performance.

Proof of Concept: The contract uses dynamic arrays for storing candidates and rankings:

```
    address[] private VOTERS;
    address[] private s_candidateList;
```

Recommended Mitigation:

- *Use Fixed-Size Arrays or Mappings:* If the maximum number of candidates or voters is known, consider using fixed-size arrays or mappings to reduce gas costs associated with dynamic resizing.
- *Optimize Data Structures:* Consider alternative data structures that are more gas-efficient for the specific operations required by the contract.
- *Batch Processing:* For operations that involve large arrays, consider processing in batches to manage gas costs and performance.

L-03. Hardcoded Constants Without Explanation

Description: The `RankedChoice` contract includes hardcoded constants, such as `MAX_CANDIDATES` and `i_presidentalDuration`, without any accompanying explanation or rationale. This can make it difficult for others to understand the reasoning behind these values and adjust them if necessary.

Impact:

- *Lack of Clarity:* Without explanations, it is unclear why certain values were chosen, which can lead to confusion or incorrect assumptions about the contract's behavior.
- *Difficulty in Maintenance:* Future developers may struggle to modify or extend the contract if they do not understand the purpose of these constants.
- *Potential Misconfiguration:* Hardcoded values may not be suitable for all use cases, and without clear documentation, they may be incorrectly configured.

Proof of Concept: The contract includes hardcoded constants without explanation:

```
    uint256 private constant MAX_CANDIDATES = 10;
    uint256 private immutable i_presidentalDuration = 1460 days;
```

Recommended Mitigation:

- *Add Comments to Explain Constants:* Provide comments explaining the purpose and rationale behind each constant value.
- *Consider Configurability:* If appropriate, allow these values to be configurable at deployment or through governance mechanisms to accommodate different use cases.
- *Document Assumptions:* Clearly document any assumptions or constraints related to these constants to guide future developers and users.

L-04. Potential Redundancy in `_isInArray` Function

Description: The `_isInArray` function is used to check if an address exists within an array. This function performs a linear search, which can be inefficient, especially if the array is large. Additionally, if the array is frequently accessed or modified, this approach can lead to redundant operations and increased gas costs.

Impact:

- *Inefficiency:* The linear search approach can lead to high gas costs and slower execution times, particularly with large arrays.
- *Redundancy:* Repeatedly checking for membership in an array using this method can be redundant if more efficient data structures are available.

Proof of Concept: The `_isInArray` function performs a linear search:

```
    function _isInArray(address[] memory array, address someAddress) internal pure
returns (bool) {
        for (uint256 i = 0; i < array.length; i++) {
            if (array[i] == someAddress) {
                return true;
            }
        }
        return false;
    }
```

Recommended Mitigation:

- *Use Mappings for Membership Checks:* Replace arrays with mappings where possible to allow for constant time complexity membership checks.
- *Optimize Data Structures:* Consider alternative data structures that provide more efficient membership testing, especially for frequently accessed data.
- *Minimize Array Usage:* Use arrays only when necessary and ensure they are kept as small as possible to reduce the impact of linear searches.

L-05. Lack of Input Size Limitation for `rankCandidatesBySig`

Description: The `rankCandidatesBySig` function allows users to submit an array of candidate addresses without any explicit limitation on the size of this input. This could potentially lead to excessively large inputs, which can result in high gas consumption and even denial of service if the transaction runs out of gas.

Impact:

- *Excessive Gas Costs:* Large inputs can lead to high gas costs, making transactions expensive for users.
- *Denial of Service (DoS):* If the input size is too large, it could cause the transaction to run out of gas, preventing the function from executing successfully and potentially disrupting the voting process.

Proof of Concept: The function currently does not impose a size limit on orderedCandidates:

```
    function rankCandidatesBySig(
        address[] memory orderedCandidates,
        bytes memory signature
```

```
    ) external {
        // No size check on orderedCandidates
        ...
    }
```

Recommended Mitigation:

- *Implement Input Size Checks:* Enforce a maximum size for the orderedCandidates array to prevent excessively large inputs.
- *Validate Input Length:* Ensure that the length of the input array is within reasonable bounds before processing it.
- *Optimize Gas Usage:* Consider optimizing the function to handle inputs more efficiently, reducing the risk of running out of gas.

L-06. Use of `block.timestamp` for Time Calculations

Description: The contract uses `block.timestamp` to manage time-based logic, such as determining when the voting period ends. While `block.timestamp` is generally reliable, it can be manipulated slightly by miners, which might affect time-sensitive operations.

Impact:

- *Minor Timestamp Manipulation:* Miners can manipulate the timestamp by a few seconds, which could potentially affect the timing of critical operations, such as the start or end of a voting period.
- *Potential Exploitation:* In scenarios where precise timing is crucial, this manipulation could be exploited to gain a slight advantage.

Proof of Concept: The contract uses block.timestamp in the selectPresident function to check if the voting period has ended:

```
    if (block.timestamp - s_previousVoteEndTimeStamp <= i_presidentalDuration) {
        revert RankedChoice__NotTimeToVote();
    }
```

Recommended Mitigation:

- *Allow for Time Buffer:* Implement a buffer period to account for potential minor timestamp manipulation, ensuring that critical operations are not affected by small changes.
- *Use Block Number for Critical Timing:* For operations where precise timing is crucial, consider using block numbers instead of timestamps, as they are less susceptible to manipulation.
- *Document Assumptions:* Clearly document any assumptions related to time calculations and the potential impact of timestamp manipulation.