PresidentElector Security Audit Report by MK (14th DEC 2024)

High Risk Findings

[H-1] Improper Vote Timing Logic Results in Immediate Presidential Election

Description: A high-severity vulnerability was discovered in the `RankedChoice` smart contract, where the `selectPresident` function can be executed immediately after deployment. This issue allows a president to be chosen without adhering to the required election duration (`i_presidentalDuration`), which undermines the integrity of the election process.

Impact: This vulnerability allows a voter to bypass the intended election cycle and select a president immediately after contract deployment. A malicious actor could take advantage of this by electing themselves or another favored candidate right after deployment, disrupting the intended governance process and potentially centralizing control of the system.

Proof of Concept:

1. The vulnerability stems from the initialization of the `s_previousVoteEndTimeStamp`, which is set to zero upon contract deployment. In test environments or simulated blockchains, where `block.timestamp` starts at 0 or 1, this allows an immediate call to `selectPresident` without waiting for the required election duration.

2. On mainnet or other live networks, `block.timestamp` will reflect the actual time since the Unix epoch, meaning the timestamp at deployment will not be zero but a large value corresponding to the time of deployment. However, the logic still allows for election timing manipulation.

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

3. The contract logic still allows immediate president selection after deployment due to the `s_previousVoteEndTimeStamp` being initialized to zero, leading to an incorrect comparison.

4. Write a test:

```
function testVoteAndSelect() public {
        orderedCandidates = [candidates[0]];
        vm.prank(voters[0]);
        rankedChoice.rankCandidates(orderedCandidates);
        console.log(
            "getPreviousVoteEndTimeStam in test: ",
            rankedChoice.getPreviousVoteEndTimeStam()
        ); // output 0
        console.log("block.timestamp in test: ", block.timestamp); // output 1

        // set timestamp to be accurated with mainnet:
        uint256 mainnetTimeStampAtDeployment = 1726271610;
        vm.warp(mainnetTimeStampAtDeployment);

        // a voter chooses his favorite president (probably himself with another
wallet)
        assertEq(rankedChoice.getUserCurrentVote(voters[0]), orderedCandidates);

        //inmediately call to select president
        rankedChoice.selectPresident();
        assertEq(rankedChoice.getCurrentPresident(), candidates[0]);
    }
```

5. Output log:

```
/// CONSOLE OUTPUT
[PASS] testVoteAndSelect() (gas: 689850)
Logs:
  getPreviousVoteEndTimeStam in test:  0
  block.timestamp in test:  1
```

Recommended Mitigation: To mitigate this issue, two potential approaches are suggested:

1. **Implement a Logical Delay for Voting**: Modify the contract to introduce a delay before the first election can take place, ensuring that no president can be selected immediately upon deployment. This could involve setting up logic that establishes a minimum time frame that must elapse before the first vote is allowed, providing a safeguard against premature elections.

2. **Set `s_previousVoteEndTimeStamp` to a Specific Date**: Alternatively, upon deployment, set the `s_previousVoteEndTimeStamp` to a specific timestamp that corresponds to the election cycle of the current president. This timestamp should be aligned with an appropriate time period, allowing voters to cast their votes within a reasonable timeframe without feeling rushed. By initializing this timestamp to a realistic and fair election cycle date, the integrity of the election process is preserved.

Medium Risk Findings

[M-1] Replay Attack and Signature Manipulation

Description: The RankedChoice contract is vulnerable to replay attacks due to the lack of a mechanism to prevent signature reuse in the `rankCandidatesBySig` function. This allows attackers to exploit valid signatures from previous transactions, enabling them to submit multiple votes. As a result, attackers could potentially alter the election outcome by repeatedly submitting the same vote. The absence of proper protection against replay attacks poses a significant security risk in the contract’s election process.

Impact: An attacker can potentially manipulate the voting process by re-submitting previously signed votes, thereby altering the election outcome. This vulnerability undermines the integrity of the voting process by allowing vote duplication.

Proof of Concept:

1. A voter signs a ranked list of candidates and submits it via `rankCandidatesBySig`.
2. An attacker intercepts this transaction and re-submits the same signature multiple times.
3. The contract accepts the signature repeatedly, since no nonce or `s_voteNumber` is used to invalidate reused signatures.

Recommended Mitigation:

1. Nonce Implementation: Include a nonce for each voter, ensuring that each vote submission is unique. If a signature is reused, the nonce would prevent it from being accepted again.

Example:

```
contract RankedChoice is EIP712 {
  ...
  +    mapping (address =>(mapping(bytes32 => bool)) public nonce;

 // Inside rankCandidatesBySig
 function rankCandidatesBySig(
        address[] memory orderedCandidates,
        bytes memory signature,
        uint256 nonce
 ) external {
   if(s_voterNonces[msg.sender][nonce]) revert throwCustomERROR();
   bytes32 structHash = keccak256(abi.encode(TYPEHASH, orderedCandidates, nonce));
   ...
   s_voterNonces[msg.sender][nonce] = true;
}
```

2. **Include `s_voteNumber` in Signature**: Incorporating the `s_voteNumber` (current election round) in the signature hash ensures that each election requires a new signature. This strengthens security by tying signatures to specific voting periods.

```
contract RankedChoice is EIP712 {
  ...

 // Inside rankCandidatesBySig
 function rankCandidatesBySig(
        address[] memory orderedCandidates,
        bytes memory signature,
+       uint256 s_voteNumber
 ) external {
-  bytes32 structHash = keccak256(abi.encode(TYPEHASH, orderedCandidates));
+  bytes32 structHash = keccak256(abi.encode(TYPEHASH, orderedCandidates,
s_voteNumber));
```

```
    ...
}
```

[M-2] Incorrect `TYPEHASH` definition renders the vote-with-signatures functionality unusable.

Description:

1. In the protocol's docs, it states that a big part of the protocol's functionality is that voters can let others spend gas for they're votes by using signatures. The following function is in charge of doing so:

```
function rankCandidatesBySig(address[] memory orderedCandidates, bytes memory
signature) external {
        bytes32 structHash = keccak256(abi.encode(TYPEHASH, orderedCandidates));
        bytes32 hash = _hashTypedDataV4(structHash);
        address signer = ECDSA.recover(hash, signature); // @audit
        _rankCandidates(orderedCandidates, signer);
}
```

2. The function uses a variable called \`TYPEHASH\` that contains the function's signature and is encoded to create the hash eventually. However, in the \`TYPEHASH\` declaration the function's signature is wrong:

```
bytes32 public constant TYPEHASH = keccak256("rankCandidates(uint256[])");
```

3. The function takes an array of addresses not uint256!

Impact: In the `rankCandidatesBySig` function the `hash` will be encoded with the wrong input parameters for the function, which will revert every transaction made with the signatures. Which means that an important capability in the protocol will not work!

Proof of Concept:

Here is a step by step of a situation of a voter trying to vote with some other user to pay the gas for them:

1. A voter will make an order list of they're candidates.

2. That voter will make a signature constructed with the function signature and the actual list they made.

3. The gas payer will now pass on the actual list and the signature.

4. The `rankCandidatesBySig` will create the hash using the wrong TYPEHASH.

5. The ECDSA contract will revert

Recommended Mitigation:

1. Fix the TYPEHASH to the correct input parameters:

```
+ bytes32 public constant TYPEHASH = keccak256("rankCandidates(address[])");
- bytes32 public constant TYPEHASH = keccak256("rankCandidates(uint256[])");
```

[M-3] No Upper Bound for Candidate Votes (Out of Gas Risk)

Description: The `selectPresident` function in the contract iterates through all voters and candidates, which can cause excessive gas usage as the candidate list expands. This can lead to out-of-gas errors or inefficient execution, especially after multiple rounds of voting. As the `candidateList` grows, the function's gas consumption increases, making it more prone to failure. This vulnerability poses a scalability issue for the election process.

Impact: This could prevent the election from being completed, blocking the selection of a new president and rendering the contract unusable during such elections.

Recommended Mitigation:

1. Enforce a maximum limit on the number of candidates that can participate in any single election. This ensures that the candidate list remains manageable and avoids excessive gas usage.

```
uint256 private constant MAX_CANDIDATES = 10;

function selectPresident() external {
    if (s_candidateList.length > MAX_CANDIDATES) {
        revert RankedChoice__InvalidInput();
    }

    // so....
}
```

[M-4] Immutable Voters in `RankedChoice` Contract

Description: The `RankedChoice` smart contract implements a ranked voting system where voters rank candidates and the president is chosen through a series of eliminations. The list of voters (`VOTERS`) is immutable, meaning it cannot be altered after contract deployment. This feature ensures that only predefined voters can participate in elections, enhancing fairness and preventing external interference in the voting process. However, this design choice can lead to potential issues regarding voter flexibility and adaptability over time.

Impact:

1. The immutability of the voters list can have the following consequences:
2. The inability to modify the voters list reduces the flexibility of the contract, particularly in situations where the voter base changes over time.
3. New eligible voters cannot be added, which may lead to disenfranchisement if the voter base evolves after the contract's deployment.
4. If a voter becomes ineligible after the contract is deployed, there is no mechanism to remove them, which could allow them to participate in elections unfairly.

5. In dynamic environments where voter eligibility changes, this inflexibility could result in outdated or inaccurate elections.

Proof of Concept:

1. The voters list is set during the contract's deployment and is stored in the `VOTERS` array. Once this list is initialized in the constructor, there is no functionality provided to add, remove, or update voters

```
    constructor(address[] memory voters) EIP712("RankedChoice", "1") {
>>      VOTERS = voters;
        i_presidentalDuration = 1460 days;
        s_currentPresident = msg.sender;
        s_voteNumber = 0;
    }
```

2. In this section, the `VOTERS` array is initialized when the contract is deployed and remains unchanged throughout the contract's lifecycle.

Recommended Mitigation:

1. Introduce functions to add, remove or update voters, with appropriate access controls to prevent unauthorized changes.

[M-5] Voters can change their vote

Description: The `RankedChoice` smart contract allows voters to rank candidates for the election of a president. However, a potential vulnerability exists due to the absence of time-based restrictions on when voters can modify their votes. This could allow last-minute vote changes, potentially leading to manipulative voting behavior and unfair election outcomes.

Impact: The absence of time-based restrictions could lead to voters adjusting their rankings at the last moment, creating an unfair advantage for those with knowledge of others' rankings, late-stage vote changes could drastically shift the election results, rendering earlier votes less impactful and also decreased election integrity where without a clear voting deadline, the election process may appear less transparent and more susceptible to tactical voting.

Proof of Concept: In the current implementation of the `RankedChoice` smart contract, voters can change their rankings at any time before the `selectPresident` function is called. Since there is no deadline or lock-in period for finalizing votes, this means voters have the flexibility to alter their choices as late as possible in the election process. This introduces an element of unpredictability and opens the system to strategic manipulation, where voters may wait for the results of others' rankings and adjust their votes accordingly. Specifically, the contract lacks any function or mechanism that prevents vote changes after a certain period. The system only enforces the selection of the president after the `presidentalDuration` has passed but doesn't limit voter activity within that time window.

Recommended Mitigation: Introduce a deadline or time-based restriction, such as a predefined voting period. Once the deadline is reached, votes should be locked, and no further changes to rankings should be allowed or modify the `selectPresident` function to only accept rankings that were finalized before the voting deadline. This ensures that the election results are based on the votes that were submitted within the allowed time window.

[M-6] No checks for time constraints to prevent voters from submitting or modifying votes after the voting period had ended

Description: The `RankedChoice` contract does not enforce any time restriction on when voters can submit their candidate rankings. This omission allows users to vote at any time, including long after the election period has ended, which could lead to invalid or manipulated election outcomes.

Impact:

1. Voters may unintentionally or maliciously submit votes after the election should have ended, leading to inaccurate results.

2. Attackers could exploit the lack of time restriction to manipulate the voting process by submitting votes outside of the official voting window, especially after assessing preliminary results.

3. Without a clear voting window, the legitimacy of election results could be questioned, undermining the system's integrity.

Proof of Concept:

The vulnerability arises from the absence of a clear voting window mechanism. Voters can submit their votes through the `rankCandidates` and `rankCandidatesBySig` functions at any time, even after the election has ended. Without restricting the voting period:

1. Voters may cast their vote outside the intended voting window.
2. Malicious actors could influence the election by submitting votes after seeing preliminary results, manipulating the outcome.
3. There is no mechanism to ensure that votes are only submitted during the active voting period.

Recommended Mitigation:

1. Ensure that the start and end timestamps are configured when initiating the election, providing a clear voting window.

2. Introduce a voting window by adding a start and end timestamp to the voting period. Ensure votes are only accepted within this period.

Example Solution:

```
uint256 private s_votingStartTime;
uint256 private s_votingEndTime;

modifier onlyDuringVoting() {
    require(block.timestamp >= s_votingStartTime && block.timestamp <=
s_votingEndTime, "Voting is not allowed at this time.");
    _;
}
```

```
function rankCandidates(address[] memory orderedCandidates) external
onlyDuringVoting {
    _rankCandidates(orderedCandidates, msg.sender);
}

function rankCandidatesBySig(address[] memory orderedCandidates, bytes memory
signature) external onlyDuringVoting {
    bytes32 structHash = keccak256(abi.encode(TYPEHASH, orderedCandidates));
    bytes32 hash = _hashTypedDataV4(structHash);
    address signer = ECDSA.recover(hash, signature);
    _rankCandidates(orderedCandidates, signer);
}
```