Security Audit Report: RankedChoice.sol

Project: President Elector - Ranked Choice Voting
Scope: `src/RankedChoice.sol`
Solidity Version: 0.8.24
Date: 2026-04-02
Auditor: Khant Wai Yan Aung

Summary

Severity	Count
High	4
Medium	3
Low	3
Info	4
Gas	3
Total	**18**

The `RankedChoice.sol` contract implements an on-chain ranked choice voting system for electing a president. The audit found critical issues in the EIP-712 meta-transaction feature (broken signature verification, missing replay protection), a timing vulnerability allowing immediate elections after deployment, and several medium/low issues around input validation and front-running.

Findings Overview

ID	Title	Severity
H-01	`selectPresident()` callable immediately after deployment, bypassing 4-year term	High
H-02	TYPEHASH uses wrong parameter type `uint256[]` instead of `address[]`	High
H-03	EIP-712 dynamic array encoding is incorrect	High
H-04	`rankCandidatesBySig` has no nonce/replay protection	High
M-01	No duplicate voter check in constructor allows double voting power	Medium
M-02	No duplicate candidate validation in `rankCandidates`	Medium
M-03	Unlimited vote overwrites enable front-running attacks	Medium
L-01	No `address(0)` validation for candidates	Low
L-02	No event emissions for critical state changes	Low
L-03	Constructor does not validate non-empty voters array	Low
I-01	`VOTERS` uses constant naming convention but is mutable	Info
I-02	`_isInArray` O(n) in nested loops - gas DoS risk	Info

I-03	Unbounded recursion in `_selectPresidentRecursive` can exceed call stack	Info
I-04	`block.timestamp` dependency in election timing (Slither)	Info
G-01	Storage array `.length` read in every loop iteration (Slither + Aderyn)	Gas
G-02	Costly SSTORE operations inside loops (Aderyn)	Gas
G-03	`s_candidateList` reset uses `new address[](0)` instead of `delete`	Gas

Detailed Findings

High

[H-01] `selectPresident()` callable immediately after deployment, bypassing the 4-year presidential term

Location: `src/RankedChoice.sol:61-66`

Description:
`s_previousVoteEndTimeStamp` is never initialized in the constructor and defaults to `0`. The timing guard:

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

Computes `block.timestamp - 0`. On any modern chain, `block.timestamp` (e.g., ~1.7 billion for 2024) is far greater than `1460 days` (126,144,000 seconds). The check always passes immediately after deployment.

Impact:

- The deployer's intended 4-year initial presidential term is completely bypassed.
- An attacker who is an approved voter can submit a single vote and call `selectPresident()` in the same block as deployment, electing their preferred candidate before any other voter participates.
- The entire initial election process can be hijacked.

Proof of Concept:

```
// 1. Contract is deployed, deployer is president
// 2. Attacker (an approved voter) sees deployment tx in mempool
// 3. Attacker submits: rankCandidates([attackerChoice])
// 4. Attacker calls: selectPresident()
// 5. attackerChoice is now president — all in the same block
```

Recommended Fix:
Initialize `s_previousVoteEndTimeStamp` to `block.timestamp` in the constructor:

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

```
    s_previousVoteEndTimeStamp = block.timestamp; // <-- Fix
    s_voteNumber = 0;
}
```

[H-02] TYPEHASH uses wrong parameter type `uint256[]` instead of `address[]`

Location: `src/RankedChoice.sol:23`

Description:
The EIP-712 TYPEHASH is:

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

But the actual function parameter is `address[] memory orderedCandidates`. Per EIP-712, the type string must match the actual Solidity types. Any off-chain EIP-712 signer (MetaMask, ethers.js, etc.) using the correct `address[]` type will produce a different struct hash, causing `ECDSA.recover` to return a wrong address.

Impact:
`rankCandidatesBySig()` will always fail. The meta-transaction voting feature advertised in the README ("Voters can also sign an un-sent transaction so others can spend the gas to cast their vote") is non-functional.

Recommended Fix:

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

[H-03] EIP-712 dynamic array encoding is incorrect

Location: `src/RankedChoice.sol:54`

Description:
The struct hash is computed as:

```
bytes32 structHash = keccak256(abi.encode(TYPEHASH, orderedCandidates));
```

Per the EIP-712 specification, dynamic types (including arrays) must be encoded as the `keccak256` of their concatenated encoded elements. Using `abi.encode` on a dynamic array includes the ABI offset pointer and length prefix, which does NOT comply with EIP-712.

Specifically, `abi.encode(TYPEHASH, orderedCandidates)` produces:

```
[TYPEHASH][offset=64][...][length][elem0][elem1]...
```

But EIP-712 requires:

```
[TYPEHASH][keccak256(abi.encodePacked(elem0, elem1, ...))]
```

Impact:
Combined with H-02, this makes signature-based voting completely incompatible with any standard EIP-712 signing library. Even if the TYPEHASH were corrected, signatures would still fail.

Recommended Fix:

```
bytes32 structHash = keccak256(
    abi.encode(TYPEHASH, keccak256(abi.encodePacked(orderedCandidates)))
);
```

[H-04] `rankCandidatesBySig` has no nonce or replay protection

Location: `src/RankedChoice.sol:50-58`

Description:
The signed message only covers `TYPEHASH` and `orderedCandidates`:

```
bytes32 structHash = keccak256(abi.encode(TYPEHASH, orderedCandidates));
bytes32 hash = _hashTypedDataV4(structHash);
address signer = ECDSA.recover(hash, signature);
```

It does NOT include `s_voteNumber`, a nonce, or any expiry. This enables:

1. **Cross-election replay:** A signature from election N can be submitted in election N+1, N+2, etc., casting votes for the signer without their knowledge or consent in future elections.
2. **Vote overwrite replay:** If a voter signs a vote via `rankCandidatesBySig`, then later changes their mind and calls `rankCandidates()` directly, an attacker holding the old signature can replay it to overwrite the voter's updated choice (since `_rankCandidates` always overwrites).

Impact:
Attackers can collect signatures from any election and replay them indefinitely to manipulate future elections. A voter who signed once has their voting autonomy permanently compromised for all future elections.

Recommended Fix:
Include `s_voteNumber` and a per-voter nonce in the signed struct:

```
bytes32 public constant TYPEHASH = keccak256(
    "rankCandidates(address[] orderedCandidates,uint256 voteNumber,uint256 nonce)"
);
```

Use OpenZeppelin's `Nonces` contract:

```
bytes32 structHash = keccak256(
    abi.encode(
        TYPEHASH,
        keccak256(abi.encodePacked(orderedCandidates)),
        s_voteNumber,
        _useNonce(signer)
    )
);
```

Medium

[M-01] No duplicate voter check in constructor allows double voting power

Location: `src/RankedChoice.sol:39-44`

Description:
The constructor stores the `voters` array directly without checking for duplicates:

```
constructor(address[] memory voters) EIP712("RankedChoice", "1") {
    VOTERS = voters; // No uniqueness check
}
```

If the same address appears multiple times, that voter is iterated multiple times in both `selectPresident()` and `_selectPresidentRecursive()`:

```
for (uint256 i = 0; i < VOTERS.length; i++) { // duplicate voter counted N times
```

Impact:
A duplicate voter has proportionally more voting power. If intentional by the deployer, this is a backdoor. If accidental, it corrupts election integrity.

Recommended Fix:
Validate uniqueness in the constructor:

```
mapping(address => bool) used;
for (uint256 i = 0; i < voters.length; i++) {
    if (used[voters[i]]) revert RankedChoice__InvalidInput();
    used[voters[i]] = true;
}
```

[M-02] No duplicate candidate validation in `rankCandidates`

Location: `src/RankedChoice.sol:159-173`

Description:
A voter can submit rankings with duplicate candidates, e.g., `[A, A, B]`. During ranked-choice elimination in `_selectPresidentRecursive`:

```
for (uint256 j = 0; j < s_rankings[VOTERS[i]][s_voteNumber].length; j++) {
    address candidate = s_rankings[VOTERS[i]][s_voteNumber][j];
    if (_isInArray(candidateList, candidate)) {
        s_candidateVotesByRound[candidate][...] += 1;
        break;
    }
}
```

The first occurrence of `A` counts as a vote. When `A` is eliminated, the voter's next choice is the second `A` (already eliminated), which is skipped, wasting their second-ranked choice. Their effective ranking becomes `[A, B]` instead of having a meaningful second choice.

Impact:
Voters can accidentally or intentionally submit malformed rankings that reduce their effective number of ranked choices, distorting election outcomes.

Recommended Fix:
Add duplicate candidate detection in `_rankCandidates`.

[M-03] Unlimited vote overwrites enable front-running attacks

Location: `src/RankedChoice.sol:172`

Description:
`_rankCandidates` unconditionally overwrites the voter's previous ranking:

```
s_rankings[voter][s_voteNumber] = orderedCandidates;
```

Combined with the fact that all votes are publicly visible on-chain and there is no voting deadline or commit-reveal scheme:

1. A voter can wait until just before `selectPresident()` is called
2. Inspect all other voters' rankings on-chain
3. Strategically submit (or change) their vote to manipulate the outcome
4. Use MEV to front-run the `selectPresident()` transaction

Impact:
Strategic vote manipulation. The last voter to act has full information about all other votes and can optimize their ranking for maximum impact.

Recommended Fix:
Implement a commit-reveal scheme or a voting deadline that locks rankings before `selectPresident()` can be called.

Low

[L-01] No `address(0)` validation for candidates

Location: `src/RankedChoice.sol:159-173`

Description:
Voters can include `address(0)` in their ranked candidates. If `address(0)` wins the election, `s_currentPresident` is set to the zero address, which most protocols and integrations interpret as "no owner" or "burned."

Recommended Fix:

```
for (uint256 i = 0; i < orderedCandidates.length; i++) {
    if (orderedCandidates[i] == address(0)) revert RankedChoice__InvalidInput();
}
```

[L-02] No event emissions for critical state changes

Location: Entire contract

Description:
The contract emits zero events. Key state changes that should be logged:

- Vote submissions (`rankCandidates` , `rankCandidatesBySig`)
- President elections (`selectPresident`)
- Vote number increments

Impact:
Off-chain monitoring, indexing (The Graph, etc.), UI updates, and election transparency are all impossible without parsing internal state via `eth_call` .

Recommended Fix:

```
event VoteCast(address indexed voter, uint256 indexed voteNumber, address[] orderedCandidates);
event PresidentElected(address indexed president, uint256 indexed voteNumber);
```

[L-03] Constructor does not validate non-empty voters array

Location: `src/RankedChoice.sol:39`

Description:
If an empty `voters` array is passed to the constructor, the contract deploys successfully but becomes permanently non-functional:

- `rankCandidates()` always reverts (`_isInArray` returns false for any caller)
- `selectPresident()` will always revert (no candidates can ever be added)
- The deployer remains president forever with no mechanism for removal

Recommended Fix:

```
if (voters.length == 0) revert RankedChoice__InvalidInput();
```

Informational

[I-01] `VOTERS` uses constant naming convention but is a mutable storage variable

Location: `src/RankedChoice.sol:27`

Description:
`VOTERS` is named in `ALL_CAPS` , which by Solidity convention indicates a `constant` or `immutable` variable. The comment notes that "Solidity doesn't support constant reference types," but the naming is misleading during code review and could cause auditors or developers to incorrectly assume immutability guarantees.

Recommendation: Consider renaming to `s_voters` to follow the storage variable convention used elsewhere in the contract.

[I-02] `_isInArray` is O(n) and called in nested loops - potential gas DoS

Location: `src/RankedChoice.sol:175-185`

Description:
`_isInArray` performs a linear scan and is called inside nested loops in `selectPresident()` and `_selectPresidentRecursive()` . The overall complexity is approximately O(voters * candidates^2 * rounds). For large voter sets or many candidates, `selectPresident()` could exceed the block gas limit, making it impossible to complete an election.

Recommendation: Use `mapping(address => bool)` for O(1) candidate and voter lookups.

[I-03] Unbounded recursion in `_selectPresidentRecursive` can exceed call stack depth

Location: `src/RankedChoice.sol:98-157`

Description:
Each voter can nominate up to `MAX_CANDIDATES` (10) unique candidates. With N voters, there could be up to `10 * N`

unique candidates total. `_selectPresidentRecursive` recurses once per eliminated candidate (depth = total candidates - 1). Solidity has a call stack limit of ~1024 frames.

With more than ~1024 unique candidates (achievable with ~103 voters each nominating 10 unique addresses), the recursion exceeds the stack depth and `selectPresident()` permanently reverts -- making it impossible to ever elect a new president.

Recommendation: Replace recursion with an iterative loop.

[I-04] `block.timestamp` dependency in election timing (Slither)

Location: `src/RankedChoice.sol:62-63`

Detected by: Slither (`timestamp` detector)

Description:
The election timing relies on `block.timestamp`, which miners/validators can manipulate within a small range (~15 seconds). While this is low-impact for a 4-year duration, it could theoretically allow a validator to trigger or prevent an election at a boundary.

```
block.timestamp - s_previousVoteEndTimeStamp <= i_presidentalDuration
```

Recommendation: Acceptable for the 4-year scale, but worth noting for awareness.

Gas Optimization

[G-01] Storage array `.length` read in every loop iteration

Location: `src/RankedChoice.sol:68, 107`

Detected by: Slither (`cache-array-length`), Aderyn (L-4)

Description:
`VOTERS.length` is read from storage on every iteration of the loop. Storage reads (SLOAD) cost ~100 gas each. With 100 voters, this wastes ~10,000 gas per loop.

```
for (uint256 i = 0; i < VOTERS.length; i++) { // SLOAD every iteration
```

Recommended Fix:

```
uint256 votersLength = VOTERS.length;
for (uint256 i = 0; i < votersLength; i++) {
```

[G-02] Costly SSTORE operations inside loops

Location: `src/RankedChoice.sol:68-77, 107-123`

Detected by: Aderyn (L-1)

Description:
In `selectPresident()`, `s_candidateList.push()` (SSTORE) is called inside a nested loop. Each storage write costs ~20,000 gas for new slots. Building the candidate list in memory first, then writing to storage once, would be significantly cheaper.

Similarly, `s_candidateVotesByRound` is updated with SSTORE inside nested loops in `_selectPresidentRecursive`.

Recommended Fix:
Build candidate lists in memory arrays first, then batch-write to storage.

[G-03] `s_candidateList` reset uses `new address[](0)` instead of `delete`

Location: `src/RankedChoice.sol:90`

Description:

```
s_candidateList = new address[](0);
```

Using `delete s_candidateList` is more gas-efficient and idiomatic for clearing a storage array, as it triggers gas refunds for freed storage slots.

Recommended Fix:

```
delete s_candidateList;
```