CANTINA

Rhinestone+Biconomy: Smartsessions External Policies

Security Review

Cantina Managed review by:
Riley Holterhus, Lead Security Researcher
Blockdev, Security Researcher
Chinmay Farkya, Associate Security Researcher

October 8, 2024

Contents

1 Introduction

1.1 About Cantina

Cantina is a security services marketplace that connects top security researchers and solutions with clients. Learn more at cantina.xyz

1.2 Disclaimer

Cantina Managed provides a detailed evaluation of the security posture of the code at a particular moment based on the information available at the time of the review. While Cantina Managed endeavors to identify and disclose all potential security issues, it cannot guarantee that every vulnerability will be detected or that the code will be entirely secure against all possible attacks. The assessment is conducted based on the specific commit and version of the code provided. Any subsequent modifications to the code may introduce new vulnerabilities that were absent during the initial review. Therefore, any changes made to the code require a new security review to ensure that the code remains secure. Please be advised that the Cantina Managed security review is not a replacement for continuous security measures such as penetration testing, vulnerability scanning, and regular code reviews.

1.3 Risk assessment

Severity	Description
Critical	*Must* fix as soon as possible (if already deployed).
High	Leads to a loss of a significant portion (>10%) of assets in the protocol, or significant harm to a majority of users.
Medium	Global losses <10% or losses to only a subset of users, but still unacceptable.
Low	Losses will be annoying but bearable. Applies to things like griefing attacks that can be easily repaired or even gas inefficiencies.
Gas Optimization	Suggestions around gas saving practices.
Informational	Suggestions around best practices or readability.

1.3.1 Severity Classification

The severity of security issues found during the security review is categorized based on the above table. Critical findings have a high likelihood of being exploited and must be addressed immediately. High findings are almost certain to occur, easy to perform, or not easy but highly incentivized thus must be fixed as soon as possible.

Medium findings are conditionally possible or incentivized but are still relatively likely to occur and should be addressed. Low findings a rare combination of circumstances to exploit, or offer little to no incentive to exploit but are recommended to be addressed.

Lastly, some findings might represent objective improvements that should be addressed but do not impact the project's overall security (Gas and Informational findings).

2 Security Review Summary

Rhinestone is the leading ERC-7579 modular smart account infrastructure platform. We provide tools and services that help developers build, deploy, and manage the smart account modules that will power the next wave of powerful onchain products with seamless UX.

From Aug 26th to Sep 3rd the Cantina team conducted a review of rhinestone-smartsessions on commit hash a6fc609c. In total, the team identified **3** issues in the following risk categories:

- Critical Risk: 1
- High Risk: 1
- Medium Risk: 0
- Low Risk: 1
- Gas Optimizations: 0
- Informational: 0

The present report is one of two from the review, focusing specifically on findings related to the external policy contracts rather than the core codebase.

3 Findings

3.1 Critical Risk

3.1.1 `UniActionPolicy` does not persist usages

Severity: Critical Risk

Context: *(No context files were provided by the reviewer)*

Description: The `UniActionPolicy` is an action policy that inspects certain sections of calldata and enforces limits on the inspected values. This policy also allows thresholds to be set and gradually filled over time, so that each call contributes to the total usage across all calls:

```
/**
 * @title UniActionPolicy: Universal Action Policy
 * @dev A policy that allows defining custom rules for actions based on function signatures.
 ...
 * Also, rules feature usage limits for arguments.
 * For example, you can limit not just max amount for a transfer,
 * but also limit the total amount to be transferred within a permission.
 ...
 */
```

However, the current implementation does not properly enforce these limits. To see this, notice that the only function that increments the `rule.usage.used` value is a `view` function which is not used in a way that persists the change:

```
function checkAction(/* ... */) external returns (uint256) {
    // ...
    for (uint256 i = 0; i < length; i++) {
        if (!config.paramRules.rules[i].check(data)) return VALIDATION_FAILED;
    }
    // ...
}

function check(ParamRule memory rule, bytes calldata data) internal view returns (bool) {
    // ...
    if (rule.isLimited) {
        if (rule.usage.used + uint256(param) > rule.usage.limit) {
            return false;
        }
        rule.usage.used += uint256(param);
    }
    // ...
}
```

As a result, usage amounts will never accumulate as intended. This behavior could allow, for example, more tokens to be withdrawn from an account than was originally intended.

Recommendation: To ensure usages are persisted across multiple calls, modify the `check()` function to accept a storage variable instead of `memory` and remove the `view` modifier.

Rhinestone: Fixed in PR 76.

Cantina Managed: Verified.

3.2 High Risk

3.2.1 `ERC20SpendingLimitPolicy` doesn't clear previous data

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

Description: When a policy is disabled in the `SmartSession` system, there is no logic that is executed to clean up the policy's storage. So, to avoid issues if a policy is re-enabled with the same `configId`, policies are supposed to clear any existing storage during the `initializeWithMultiplexer()` call. This is explained in the following comment from the code:

```
/**
 * Disables specified policies for a given permission ID and smart account.
 *
 * @dev This function removes the specified policies from the policy list and emits events for each disabled
↪  policy.
 * @notice Cleaning state on policies is not required as on enable, initializeWithMultiplexer is called which
↪  MUST
 *       overwrite the current state.
 ...
 */
```

However, in the `ERC20SpendingLimitPolicy`, the `initializeWithMultiplexer()` function does not fully implement this overwriting. The function is currently implemented as follows:

```
function initializeWithMultiplexer(address account, ConfigId configId, bytes calldata initData) external {
    (address[] memory tokens, uint256[] memory limits) = abi.decode(initData, (address[], uint256[]));

    for (uint256 i; i < tokens.length; i++) {
        address token = tokens[i];
        uint256 limit = limits[i];
        if (token == address(0)) revert InvalidTokenAddress(token);
        if (limit == 0) revert InvalidLimit(limit);
        TokenPolicyData storage $ = _getPolicy({ id: configId, userOpSender: account, token: token });
        $.spendingLimit = limit;
    }
    emit IPolicy.PolicySet(configId, msg.sender, account);
}
```

This is not fully correct because it does not reset the `alreadySpent` value in storage. The following modification could address this:

```
  function initializeWithMultiplexer(address account, ConfigId configId, bytes calldata initData) external {
      (address[] memory tokens, uint256[] memory limits) = abi.decode(initData, (address[], uint256[]));

      for (uint256 i; i < tokens.length; i++) {
          address token = tokens[i];
          uint256 limit = limits[i];
          if (token == address(0)) revert InvalidTokenAddress(token);
          if (limit == 0) revert InvalidLimit(limit);
          TokenPolicyData storage $ = _getPolicy({ id: configId, userOpSender: account, token: token });
          $.spendingLimit = limit;
+         $.alreadySpent = 0;
      }
      emit IPolicy.PolicySet(configId, msg.sender, account);
  }
```

However, with this change, there can still be problems if the newest `tokens` array does not match the `tokens` array from a previous installation. In this case, the existing `tokens` array would not have its storage overwritten, which can lead to re-enabling previous storage unintentionally.

Additionally, it can be noted that in most cases the `tokens` array is not fully necessary. This is because the `ERC20SpendingLimitPolicy` is an action policy, so the `configId` passed to `initializeWithMultiplexer()` will typically commit to a specific target address and function selector. This means there is only one token contract for which the spending limit can actually be reached. The only exception to this is the `FALLBACK_ACTIONID`, which can potentially reach multiple contracts from the same action policy. Since this one scenario exists where the `tokens` array can be useful, there can still be issues when the `ERC20SpendingLimitPolicy` is re-enabled and doesn't clear previous data.

Recommendation: Change the `ERC20SpendingLimitPolicy` function to ensure that all previous storage is cleared or overwritten. This may be difficult since there is no easy way to enumerate all the existing tokens that are in storage. Therefore, it may be worth considering reworking this part of the system, for example by never allowing a policy to re-enable a `configId` value after it has been disabled.

Rhinestone: Fixed in PR 117.

Cantina Managed: Verified. The `ERC20SpendingLimitPolicy` now contains an `EnumerableSet` to allow `initializeWithMultiplexer()` to iterate through and clear all previously initialized storage for the user.

3.3 Low Risk

3.3.1 ERC-165 logic uses selectors instead of interface ids

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

Description: When a user installs a policy in the `SmartSession` codebase, ERC-165 is used to ensure that the policy implements the expected interface. According to the ERC-165 specification, these types of queries should be based on an *interface identifier*, which is defined as the `xor` of all function selectors in the interface.

However, in the current codebase, some functions query using individual function selectors instead of an interface identifier. For example:

```
function requirePolicyType(address policy, PolicyType policyType) internal view {
    bool supportsInterface;
    if (policyType == PolicyType.USER_OP) {
        supportsInterface = IPolicy(policy).supportsInterface(IUserOpPolicy.checkUserOpPolicy.selector);
    } else if (policyType == PolicyType.ACTION) {
        supportsInterface = IPolicy(policy).supportsInterface(IActionPolicy.checkAction.selector);
    } else if (policyType == PolicyType.ERC1271) {
        supportsInterface = IPolicy(policy).supportsInterface(I1271Policy.check1271SignedAction.selector);
    } else {
        revert UnsupportedPolicy(policy);
    }
}
```

Similarly, notice that the following `supportsInterface()` implementation checks against an individual selector:

```
function supportsInterface(bytes4 interfaceID) external pure override returns (bool) {
    if (interfaceID == type(IActionPolicy).interfaceId) {
        return true;
    }
    if (interfaceID == IActionPolicy.checkAction.selector) {
        return true;
    }
}
```

Recommendation: To better align with the ERC-165 specification, consider changing the `requirePolicyType()` function and related `supportsInterface()` functions to check against interface identifiers rather than individual selectors.

For example, the relevant changes in `requirePolicyType()` could be the following:

```
  function requirePolicyType(address policy, PolicyType policyType) internal view {
      if (policyType == PolicyType.USER_OP) {
-         supportsInterface = IPolicy(policy).supportsInterface(IUserOpPolicy.checkUserOpPolicy.selector);
+         supportsInterface = IPolicy(policy).supportsInterface(type(IUserOpPolicy).interfaceId);
      } else if (policyType == PolicyType.ACTION) {
-         supportsInterface = IPolicy(policy).supportsInterface(IActionPolicy.checkAction.selector);
+         supportsInterface = IPolicy(policy).supportsInterface(type(IActionPolicy).interfaceId);
      } else if (policyType == PolicyType.ERC1271) {
-         supportsInterface = IPolicy(policy).supportsInterface(I1271Policy.check1271SignedAction.selector);
+         supportsInterface = IPolicy(policy).supportsInterface(type(I1271Policy).interfaceId);
      }
  }
```

For the `supportsInterface()` functions, the three policy contracts could be changed to:

- `ERC20SpendingLimitPolicy` and `UniActionPolicy`:

```
function supportsInterface(bytes4 interfaceID) external pure override returns (bool) {
    return (
        interfaceID == type(IERC165).interfaceId ||
        interfaceID == type(IPolicy).interfaceId ||
        interfaceID == type(IActionPolicy).interfaceId
    );
}
```

- `SudoPolicy`:

```
    function supportsInterface(bytes4 interfaceID) external pure override returns (bool) {
        return (
            interfaceID == type(IERC165).interfaceId ||
            interfaceID == type(IPolicy).interfaceId ||
            interfaceID == type(IActionPolicy).interfaceId ||
            interfaceID == type(I1271Policy).interfaceId
        );
    }
```

Rhinestone: Fixed in PR 107.

Cantina Managed: Verified.