Lens V3 Rules Security Review

Pashov Audit Group

Conducted by: Shaka, ubermensch, ast3ros, merlinboii

January 6th 2025 - February 3rd 2025

Contents

1. About Pashov Audit Group

Pashov Audit Group consists of multiple teams of some of the best smart contract security researchers in the space. Having a combined reported security vulnerabilities count of over 1000, the group strives to create the absolute very best audit journey possible - although 100% security can never be guaranteed, we do guarantee the best efforts of our experienced researchers for your blockchain protocol. Check our previous work here or reach out on Twitter @pashovkrum.

2. Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource and expertise bound effort where we try to find as many vulnerabilities as possible. We can not guarantee 100% security after the review or even if the review will find any problems with your smart contracts. Subsequent security reviews, bug bounty programs and on-chain monitoring are strongly recommended.

3. Introduction

A time-boxed security review of the **lens-protocol/lens-v3** repository was done by **Pashov Audit Group**, with a focus on the security aspects of the application's smart contracts implementation.

4. About Lens V3 Rules

Lens is an open social network where every EVM account acts as a Profile, supporting smart wallets and social features. It is built on modular primitives (Feed, Graph, Group, Namespace) that can be extended with custom Actions and Rules, enabling flexible and customizable interactions within the ecosystem.

This scope includes contracts implementing Lens Rules. These also serve as examples of how developers could build their own Rules.

5. Risk Classification

Severity	**Impact: High**	**Impact: Medium**	**Impact: Low**
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

5.1. Impact

- High - leads to a significant material loss of assets in the protocol or significantly harms a group of users.
- Medium - only a small amount of funds can be lost (such as leakage of value) or a core functionality of the protocol is affected.
- Low - can lead to any kind of unexpected behavior with some of the protocol's functionalities that's not so critical.

5.2. Likelihood

- High - attack path is possible with reasonable assumptions that mimic on-chain conditions, and the cost of the attack is relatively low compared to the amount of funds that can be stolen or lost.
- Medium - only a conditionally incentivized attack vector, but still relatively likely.
- Low - has too many or too unlikely assumptions or requires a significant stake by the attacker with little or no incentive.

5.3. Action required for severity levels

- Critical - Must fix as soon as possible (if already deployed)
- High - Must fix (before deployment if not already deployed)
- Medium - Should fix
- Low - Could fix

6. Security Assessment Summary

review commit hash - faf3765db6727e5e5e5f5cdd03430870a3cfb0cd

fixes review commit hash - ec11f802a3c8168219dc3725c798562726ef6b70

Scope

The following smart contracts were in scope of the audit:

- `OwnableMetadataBasedRule`
- `RestrictedSignersRule`
- `SimplePaymentRule`
- `TokenGatedRule`
- `TrustBasedRule`
- `GroupGatedFeedRule`
- `RestrictedSignersFeedRule`
- `SimplePaymentFeedRule`
- `TokenGatedFeedRule`
- `SimplePaymentFollowRule`
- `TokenGatedFollowRule`
- `GroupGatedGraphRule`
- `RestrictedSignersGraphRule`
- `TokenGatedGraphRule`
- `BanMemberGroupRule`
- `MembershipApprovalGroupRule`
- `SimplePaymentGroupRule`
- `TokenGatedGroupRule`
- `SimplePaymentNamespaceRule`
- `TokenGatedNamespaceRule`
- `UsernameCharsetNamespaceRule`
- `UsernameLengthNamespaceRule`
- `UsernamePricePerLengthNamespaceRule`
- `UsernameReservedNamespaceRule`
- `UsernameSimpleCharsetNamespaceRule`
- `FollowersOnlyPostRule`
- `AccountBlockingRule`

7. Executive Summary

Over the course of the security review, Shaka, ubermensch, ast3ros, merlinboii engaged with Avara to review Lens V3 Rules. In this period of time a total of **9** issues were uncovered.

Protocol Summary

Protocol Name	Lens V3 Rules
Repository	https://github.com/lens-protocol/lens-v3
Date	January 6th 2025 - February 3rd 2025
Protocol Type	Social Network

Findings Count

Severity	**Amount**
Medium	3
Low	6
Total Findings	**9**

Summary of Findings

ID	Title	Severity	Status
[M-01]	Incorrect unicode handling in username character set restrictions leads to bypasses	Medium	Acknowledged
[M-02]	Rule changes signature verification fails for new entities rule configurations	Medium	Acknowledged
[M-03]	Missing configSalt and rule address in RestrictedSignerMessage	Medium	Acknowledged
[L-01]	Skip gate permission is not checked in group removal	Low	Resolved
[L-02]	Conflict in process removal	Low	Resolved
[L-03]	RestrictedSignersRule cannot handle smart contract wallet with multiple signatures	Low	Acknowledged
[L-04]	Order of Restrictions Causes Valid Usernames to Be Rejected	Low	Resolved
[L-05]	Redundant Signature Deadline Check	Low	Acknowledged
[L-06]	Redundant Addresses Added to Signature	Low	Acknowledged

8. Findings

8.1. Medium Findings

[M-01] Incorrect unicode handling in username character set restrictions leads to bypasses

Severity

Impact: Medium

Likelihood: Medium

Description

The `UsernameCharsetNamespaceRule._processRestrictions` function aims to validate usernames against configured character set restrictions. However, it makes a flawed assumption: it processes the username string byte by byte using `bytes(username)`, assuming each character is represented by a single byte. This assumption is incorrect for Unicode characters, which can be composed of multiple bytes. This byte-by-byte processing leads to a vulnerability when the `customAllowedCharset` parameter is used to allow multi-byte Unicode characters.

```
function _processRestrictions
      (string calldata username, CharsetRestrictions memory charsetRestrictions)
        internal
        pure
    {
        // Cannot start with a character in the cannotStartWith charset
        require(!_isInCharset(bytes(
          !_isInCharset
        )[0], charsetRestrictions.cannotStartWith
        // Check if the username contains only allowed characters
        for (uint256 i = 0; i < bytes(username).length; i++) {
            ...
            } else if (bytes
              (charsetRestrictions.customAllowedCharset).length > 0) {
                require(_isInCharset(
                  _isInCharset

                ), Errors.NotAllowed(
            ...
        }
    }
```

Let's consider a scenario:

- A unicode character `unicode"é"` is allowed as a custom allowed character (so café is an allowed name).
- In UTF-8 encoding, "é" is represented by two bytes: `0xC3 0xA9`.
- A malicious user could then attempt to register usernames composed of individual bytes from this multi-byte character such as `0xC3C3C3C3` or `0xA9A9A9A9`
- Because `_processRestrictions` iterates byte-by-byte, and `_isInCharset` checks byte-by-byte, each individual byte (`0xC3` or `0xA9`) in these crafted usernames would be incorrectly identified as being within the `customAllowedCharset` (because `0xC3` and `0xA9` are bytes that are part of the allowed "é" character).
- It would incorrectly allow these "invalid" usernames, even though they do not represent valid sequences of the intended Unicode character "é" or other valid characters according to the rule's intent. Users can bypass the intended character set restrictions.

Recommendations

- Don't allow Unicode in the `customAllowedCharset` OR
- Iterate over the username string character by character.

[M-02] Rule changes signature verification fails for new entities rule configurations

Severity

Impact: Medium

Likelihood: Medium

Description

In the `RuleBasedPrimitive._changeRules()` function, when processing rule changes that include configuration updates, the function generates a new `configSalt` for the rule change if the provided `configSalt` is `bytes32(0x00)` (`@2>`), indicating a new configuration.

However, the new `configSalt` is passed to signature verification for the primitive that enforce the `RestrictedSignersRule` (`@3>`) instead of the original data because assigning `RuleChange memory ruleChange` from memory to memory (`@1>`), which creates a reference rather than a copy. As a result, modifications to `ruleChange.configSalt` affect the original memory instance.

```
//File: contracts/core/base/RuleBasedPrimitive.sol

function _changeRules(
    RulesStorage storage rulesStorage,
    uint256 entityId,
@>   RuleChange[] memory ruleChanges,
    ...
) private {
    _beforeChangeRules(entityId, ruleChanges);
    for (uint256 i = 0; i < ruleChanges.length; i++) {
@>      RuleChange memory ruleChange = ruleChanges[i]; //@audit memory pointer
        if (ruleChange.configurationChanges.configure) {
@1>          ruleChange.configSalt =
                _configureRule(
                  rulesStorage,
                  ruleChange,
                  entityId,
                  fn_encodeConfigureCall,
                  fn_emitConfiguredEvent
                );
        }
        --- SNIPPED ---
    }
    if (entityId == 0) {
        _validateRulesLength(rulesStorage, _supportedPrimitiveRuleSelectors());
    } else {
        _validateRulesLength(rulesStorage, _supportedEntityRuleSelectors());
@3>      _processEntityRulesChanges
  (entityId, ruleChanges, ruleChangesProcessingParams);
    }
}
```

```
//File: contracts/core/libraries/RulesLib.sol

function generateOrValidateConfigSalt(
    RulesStorage storage rulesStorage,
    address ruleAddress,
    bytes32 providedConfigSalt
) internal returns (bytes32) {
    if (providedConfigSalt == 0x00) {
@2>      return bytes32(++rulesStorage.lastConfigSaltGenerated);
    } else {
        --- SNIPPED ---
}
```

This modification can breaks signature verification `Rule.processPostRuleChanges()` for `RestrictedSignersRule` rule because the new generated `configSalt` used differs from the value originally signed.

```
//File: contracts/rules/feed/RestrictedSignersFeedRule.sol

 function processPostRuleChanges(
    bytes32 configSalt,
    uint256 postId,
    RuleChange[] calldata ruleChanges,
    KeyValue[] calldata ruleParams
) external override {
    _validateRestrictedSignerMessage({
        configSalt: configSalt,
        functionSelector: IFeedRule.processPostRuleChanges.selector,
@3>     abiEncodedFunctionParams: abi.encode
  (postId, EIP712EncodingLib.encodeForEIP712(ruleChanges)),
        signature: abi.decode(ruleParams[0].value, (EIP712Signature))
    });
}
```

Recommendations

Update the `RuleBasedPrimitive._changeRules()` function to create a new copy of the value from `ruleChanges[i]` instead of using the referenced one, ensuring that the signature verification process uses the originally signed data.

[M-03] Missing `configSalt` and `rule address` in `RestrictedSignerMessage`

Severity

Impact: High

Likelihood: Low

Description

The `RestrictedSignerMessage` is not signed with the rule's `configSalt` and the rule's `address`, creating a potential replay across internal configurations or different rule contract usage.

```
struct RestrictedSignerMessage {
    bytes4 functionSelector;
    bytes abiEncodedParams;
    uint256 nonce;
    uint256 deadline;
}
```

Each `abiEncodedParams` encodes the primitive parameters. For example:

- `RestrictedSignersFeedRule.processCreatePost()`

```
function processCreatePost(
    ...
) external override {
    _validateRestrictedSignerMessage({
        configSalt: configSalt,
        functionSelector: IFeedRule.processCreatePost.selector,
@>      abiEncodedFunctionParams: abi.encode(
            postId, EIP712EncodingLib.encodeForEIP712
              (postParams), EIP712EncodingLib.encodeForEIP712(primitiveParams)
        ),
        signature: abi.decode(ruleParams[0].value, (EIP712Signature))
    });
}
```

- `RestrictedSignersGraphRule.processFollow()`

```
function processFollow(
    ...
) external override {
    _validateRestrictedSignerMessage({
        configSalt: configSalt,
        functionSelector: IGraphRule.processFollow.selector,
@>      abiEncodedFunctionParams: abi.encode(

        ),
        signature: abi.decode(ruleParams[0].value, (EIP712Signature))
    });
}
```

Consider the following scenario of `configSalt` changing:

1. A primitive disables a previous configuration, `configSalt: 0xaa`, signers: [`owner`, `eoa_signer_1`, `eoa_signer_2`] and introduces a new one with updates on some signers, `configSalt: 0xbb`, signers: [`owner`, `contract_signer_1`, `contract_signer_2`].
2. A user obtains a valid signature signed by the `owner` from a previous operation.
3. The user replays that signature under the new configuration. Since the `owner` is still a whitelisted signer and `wasSignerNonceUsed` validation maps with `configSalt`, the signature remains valid until expiration.

Recommendations

Update the domain verification to designate `address(this)` (the rule contract) as the `verifyingContract` instead of using the primitive address (`msg.sender`) and ensure that both the `primitive address` and `configSalt` are included in the signed message.

8.2. Low Findings

[L-01] Skip gate permission is not checked in group removal

`TokenGatedGroupRule` allows accounts with the `PID__SKIP_GATE` permission to bypass the token gate check. However, in the removal process, it is not checked if the account has the `PID__SKIP_GATE` permission. This leaves the skip permission without effect, as anyone can enforce the token gate over a permissioned account by calling `processRemoval`.

Recommendations:

```
function processRemoval(
        bytes32 configSalt,
        address, /* originalMsgSender */
        address account,
        KeyValue[] calldata, /* primitiveParams */
        KeyValue[] calldata /* ruleParams */
    ) external view {
+       if (!_configuration[msg.sender][configSalt].accessControl.hasAccess
+ (account, PID__SKIP_GATE)) {
            // Anyone can kick out member of the group if they no longer hold
            // the required token balance:
            require(!_checkTokenBalance(
              !_checkTokenBalance

            ), Errors.NotAllowed(
+       }
    }
```

[L-02] Conflict in process removal

The `TokenGatedGroupRule.processRemoval` function is designed to enable permissionless removal of group members based on their token balance. The intent is that anyone (not just group admins) can trigger the removal of a member if that member no longer holds the required tokens:

```
function processRemoval(
        bytes32 configSalt,
        address, /* originalMsgSender */
        address account,
        KeyValue[] calldata, /* primitiveParams */
        KeyValue[] calldata /* ruleParams */
    ) external view {
        // Anyone can kick out member of the group if they no longer hold the
        // required token balance:
        require(!_checkTokenBalance(
          !_checkTokenBalance

        ), Errors.NotAllowed(
    }
```

However, it doesn't work because to remove a member, an account needs to have `PID__REMOVE_MEMBER` permission.

```
function removeMember(
        address account,
        KeyValue[] calldata customParams,
        RuleProcessingParams[] calldata ruleProcessingParams
    ) external override {
        _requireAccess(msg.sender, PID__REMOVE_MEMBER);
        ...
    }
```

[L-03] `RestrictedSignersRule` cannot handle smart contract wallet with multiple signatures

The `RestrictedSignersRule` does not support multisig wallet contracts. It assumes that the signature in `isValidSignature` is always 65 bytes `(abi.encodePacked(signature.r, signature.s, signature.v))`. However, the signature length can vary depending on the contract's implementation. For example, in a two-user multisig, the signature length would be 130 bytes instead of 65. Other smart contract wallets may also require different signature formats.

```
function _validateRecoveredAddress
      (bytes32 digest, EIP712Signature memory signature) private view {

        ...

        // If the expected address is a contract, check the signature there.

        if (signature.signer.code.length != 0) {

            bytes memory concatenatedSig = abi.encodePacked
              (signature.r, signature.s, signature.v);

            if (IERC1271(signature.signer).isValidSignature
              (digest, concatenatedSig) != EIP1271_MAGIC_VALUE) {

                revert Errors.InvalidSignature();

            }

        ...

    }
```

[L-04] Order of Restrictions Causes Valid Usernames to Be Rejected

The `_processRestrictions` function in `UsernameCharsetNamespaceRule` enforces character restrictions in an order that can lead to valid usernames being incorrectly rejected. Specifically, if a character type (e.g., numeric or lowercase letter) is globally disallowed via flags like `allowNumeric` or `allowLatinLowercase` but explicitly allowed in `customAllowedCharset`, the function will revert when it encounters that character:

```
for (uint256 i = 0; i < bytes(username).length; i++) {
        bytes1 char = bytes(username)[i];
        // Check disallowed chars first
        require(!_isInCharset(
          !_isInCharset

        ), Errors.NotAllowed(
        // Check allowed charsets next
        if (_isNumeric(char)) {
            require(charsetRestrictions.allowNumeric, Errors.NotAllowed());
        } else if (_isLatinLowercase(char)) {
            require(charsetRestrictions.allowLatinLowercase, Errors.NotAllowed
              ());
        } else if (_isLatinUppercase(char)) {
            require(charsetRestrictions.allowLatinUppercase, Errors.NotAllowed
              ());
        } else if (bytes(charsetRestrictions.customAllowedCharset).length > 0) {
=>            require(_isInCharset
  (char, charsetRestrictions.customAllowedCharset), Errors.NotAllowed());
        } else {
            // If not in any of the above charsets, reject
            revert Errors.NotAllowed();
        }
```

Example:

- allowing only specific letters 'abcdef':

1. `allowLatinLowercase = false` (disallowing general lowercase letters)
2. 'abcdef' is added to `customAllowedCharset`
3. the function reverts even though it should pass

This issue occurs because the function checks the global flags (e.g., `allowNumeric`, `allowLatinLowercase`) before checking `customAllowedCharset`, causing unnecessary rejections.

Recommendations:

Reorder the restriction checks to prioritize `customAllowedCharset` before the global disallow flags.

[L-05] Redundant Signature Deadline Check

The `RestrictedSignersRule` contract unnecessarily checks the signature.deadline twice:

1. In `_validateRestrictedSignerMessage`, before verifying the signer's whitelist status.
2. Again in `_validateRecoveredAddress`, where the signature is recovered.

```
function _validateRestrictedSignerMessage(
      bytes32 configSalt,
      bytes4 functionSelector,
      bytes memory abiEncodedFunctionParams,
      EIP712Signature memory signature
  ) internal {
      RestrictedSignerMessage memory message =
          RestrictedSignerMessage(
            functionSelector,
            abiEncodedFunctionParams,
            signature.nonce,
            signature.deadline
          );
=>      if (block.timestamp > signature.deadline) {
          revert Errors.Expired();
      }
      if ($rulesStorage(
        $rulesStorage

      ).wasSignerNonceUsed[signature.signer][signature.nonce]
          revert Errors.NonceUsed();
      }
      $rulesStorage(
        msg.sender,
        configSalt
      ).wasSignerNonceUsed[signature.signer][signature.nonce] = true;
      emit Lens_RestrictedSignersRule_SignerNonceUsed
        (signature.signer, signature.nonce);
      if (!$rulesStorage
        (msg.sender, configSalt).isWhitelistedSigner[signature.signer]) {
          revert Errors.WrongSigner();
      }
      bytes32 hashStruct = _calculateMessageHashStruct(message);
      bytes32 digest = _calculateDigest(hashStruct);
 =>     _validateRecoveredAddress(digest, signature);
  }
```

```
function _validateRecoveredAddress
  (bytes32 digest, EIP712Signature memory signature) private view {
=>        if (block.timestamp > signature.deadline) {
          revert Errors.Expired();
      }
```

[L-06] Redundant Addresses Added to Signature

In the `RestrictedSignersGraphRule` contract, the functions `processFollow` and `processUnfollow` validate signatures using `_validateRestrictedSignerMessage`. This function signs several parameters, including originalMsgSender and followerAccount:

```
_validateRestrictedSignerMessage({
          configSalt: configSalt,
          functionSelector: IGraphRule.processFollow.selector,
=>          abiEncodedFunctionParams: abi.encode(

          ),
          signature: abi.decode(ruleParams[0].value, (EIP712Signature))
      });
```

However, due to checks in the Graph contract, `originalMsgSender` and `followerAccount` are always the same because a user can only follow others using their own account:

```
function follow(
      address followerAccount,
      address accountToFollow,
      KeyValue[] calldata customParams,
      RuleProcessingParams[] calldata graphRulesProcessingParams,
      RuleProcessingParams[] calldata followRulesProcessingParams,
      KeyValue[] calldata extraData
  ) external virtual override returns (uint256) {
=>      require(msg.sender == followerAccount, Errors.InvalidMsgSender());
```

Since these two addresses will always be identical, including both in the signature is redundant.