shieldify

Etherspot

Credible Account Module

SECURITY REVIEW

Date: 30 June 2025

CONTENTS

1. About Shieldify

Positioned as the first hybrid Web3 Security company, Shieldify shakes things up with a unique subscription-based auditing model that entitles the customer to unlimited audits within its duration, as well as top-notch service quality thanks to a disruptive 6-layered security approach. The company works with very well-established researchers in the space and have secured multiple millions in TVL across protocols, also can audit codebases written in Solidity, Vyper, Rust, Cairo, Move and Go.

Learn more about us at shieldify.org.

2. Disclaimer

This security review does not guarantee bulletproof protection against a hack or exploit. Smart contracts are a novel technological feat with many known and unknown risks. The protocol, which this report is intended for, indemnifies Shieldify Security against any responsibility for any misbehavior, bugs, or exploits affecting the audited code during any part of the project's life cycle. It is also pivotal to acknowledge that modifications made to the audited code, including fixes for the issues described in this report, may introduce new problems and necessitate additional auditing.

3. About Etherspot - Credible Account Module

Etherspot is a top-notch Account & Chain Abstraction infrastructure designed to help developers create an unparalleled cross-chain user experience for their blockchain protocols on Ethereum and EVM-compatible chains.

The **CredibleAccountModule** is a dual-purpose ERC-7579 module that functions as both a validator and a hook for smart accounts, enabling secure session key management with resource locking and token balance validation.

This module implements session-based authentication where users can create time-limited session keys with locked token amounts. It validates user operations against session parameters and ensures sufficient unlocked token balances through pre/post execution hooks.

The **ResourceLockValidator** is a validator module for ERC-7579 smart accounts that enables secure session key management through resource locking mechanisms and Merkle proofs for batched authorizations.

This validator implements dual-mode signature verification, supporting both direct ECDSA signatures and Merkle proof-based validations. It extracts resource lock data from user operation call data and validates operations against predefined resource constraints, enabling efficient batch authorization of multiple resource locks through Merkle tree structures.

4. Risk Classification

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

4.1 Impact

- **High** – results in a significant risk for the protocol's overall well-being. Affects all or most users
- **Medium** – results in a non-critical risk for the protocol affects all or only a subset of users, but is still unacceptable
- **Low** – losses will be limited but bearable – and covers vectors similar to griefing attacks that can be easily repaired

4.2 Likelihood

- **High** – almost certain to happen and highly lucrative for execution by malicious actors
- **Medium** – still relatively likely, although only conditionally possible
- **Low** – requires a unique set of circumstances and poses non-lucrative cost-of-execution to rewards ratio for the actor

5. Security Review Summary

The security review lasted 7 days with a total of 112 hours dedicated to the audit by two researchers from the Shieldify team.

Overall, the code is well-written. The audit report contributed by identifying eight Critical, four High, two Medium and nine Low severity issues. Mostly related to session key validation, access control, and authorization gaps, all of which have been fixed.

The Etherspot team has done a great job with their test suite and provided exceptional support, and promptly implemented all of the suggested recommendations from the Shieldify researchers.

5.1 Protocol Summary

Project Name	**Etherspot – Credible Account Module**
Repository	etherspot-modular-accounts
Type of Project	Account Abstraction, ERC-7579, EIP-712
Audit Timeline	7 days
Review Commit Hash	d4774db9f544cc6f69000c55e97627f93fe7242b
Fixes Review Commit Hash	a233fcf226b3095a61fbb5244a8472e3f704f5f5

5.2 Scope

The following smart contracts were in the scope of the security review:

File	nSLOC
src/modules/validators/CredibleAccountModule.sol	453
src/modules/validators/ResourceLockValidator.sol	202
Total	**655**

6. Findings Summary

The following number of issues have been identified, sorted by their severity:

- **Critical** and **High** issues: **12**
- **Medium** issues: **2**
- **Low** issues: **9**

ID	Title	Severity	Status
[C-01]	Missing Session Key Uniqueness Validation in `ResourceLock` due to Key Data Overwrite	Critical	Fixed
[C-02]	Unauthorized Session Key Takeover via Missing Ownership Validation	Critical	Fixed
[C-03]	Wrong Wallet Validation in `disableSessionKey()` When Caller is `SESSION_KEY_DISABLER` Role	Critical	Fixed
[C-04]	Memory vs Storage Mismatch Renders Session State Update Failure in `validateUserOp()`	Critical	Fixed
[C-05]	In `CredibleAccountModule` the `validateSessionKeyParams()` Function is `public`	Critical	Fixed
[C-06]	Anyone Can Uninstall the Validator Module from Any Wallet	Critical	Fixed
[C-07]	In `ResourceLockValidator` the `validateUserOp()` Function Is Not Consuming the Signature Proof	Critical	Fixed
[C-08]	In `ResourceLockValidator`, the `validateUserOp()` Function Lacks Sufficient Checks, Allowing Draining of `ModularEtherspotWallet` Balances	Critical	Fixed
[H-01]	No Check for `userOp()` and `userOpHash()` Mismatch Nor the Validity of the Sender	High	Fixed
[H-02]	In `CredibleAccountModule` the `validateUserOp()` Function Is Not Authenticating the Sender	High	Fixed
[H-03]	The `pre/postCheck()` Is Not Authorizing `wallet` Parameter with `msg.sender`	High	Fixed
[H-04]	`ModularEtherspotWallet` Can Get DoS'ed from Uninstalling Modules	High	Fixed
[M-01]	The `_removeSessionKey()` Will Never Remove the Unbounded Session Key Array Due to Gas-Based DoS	Medium	Fixed
[M-02]	The `pre/postCheck()` Is Not Checking the Hook Initialization of the Sender	Medium	Fixed

[L-01]	Redundant Validation of `validUntil == 0` in the `enableSessionKey()` Function	Low	Fixed
[L-02]	Missing `chainID` Validation and `smartWallet` Validation in Session Key Activation	Low	Fixed
[L-03]	Redundant Selector Validation in `_validateSingleCall()`	Low	Fixed
[L-04]	Redundant Event Emission in Role Management If Role Is Already Given	Low	Fixed
[L-05]	Miner Delay of Accepting the Tx Can Revert Due To `>=` Comparison	Low	Fixed
[L-06]	Missing Timestamp Validation Allows Backdated Session Keys	Low	Fixed
[L-07]	Unbounded Tokens Addition When Enabling SessionKeys	Low	Fixed
[L-08]	There Is No Check for the `walletOwner` Existence in the `validateUserOp()` Function in `ResourceLockValidator`	Low	Fixed
[L-09]	Missing Input Length Validation in the `onInstall()` Function	Low	Fixed

7. Findings

[C-01] Missing Session Key Uniqueness Validation in `ResourceLock` due to Key Data Overwrite

Severity

Critical Risk

Description

The `enableSessionKey()` function accepts a `ResourceLock` input (`_resourceLock`) but does not validate whether the `sessionKey` is already in use for the given `msg.sender` . This leads to silent overwrites of existing session data when the same `sessionKey` is reused in multiple `ResourceLock` calls.

Location of Affected Code

File: src/modules/validators/CredibleAccountModule.sol#L136

```
function enableSessionKey(bytes calldata _resourceLock) external {
// code
  sessionData[rl.sessionKey][msg.sender] = SessionData({
      sessionKey: rl.sessionKey,
      validAfter: rl.validAfter,
      validUntil: rl.validUntil,
      live: true
  });
}
```

Impact

The main impact of this bug is silent data corruption and accounting inconsistencies. When a session key is reused, the contract overwrites existing `sessionData` without warning, leaving orphaned `lockedTokens` entries and potentially duplicating keys in `walletSessionKeys`.

This can break token claim functionality and allow attackers to disrupt operations by maliciously overwriting sessions. The system loses track of originally locked tokens while maintaining stale references, creating inconsistencies between storage mappings.

Recommendation

Add a check if it is live or not:

```
if (sessionData[rl.sessionKey][msg.sender].live) {
    revert();
}
```

Team Response

Fixed.

[C-02] Unauthorized Session Key Takeover via Missing Ownership Validation

Severity

Critical Risk

Description

The `enableSessionKey()` function fails to verify whether a session key is already registered to another wallet before assigning ownership.

This allows any attacker to:

- Submit a session key that's currently active for another user
- Overwrite `sessionKeyToWallet` mapping to point to their wallet
- Effectively hijack control of the session key

Location of Affected Code

File: src/modules/validators/CredibleAccountModule.sol#L144

```
function enableSessionKey(bytes calldata _resourceLock) external {
// code
  sessionKeyToWallet[rl.sessionKey] = msg.sender;
  emit CredibleAccountModule_SessionKeyEnabled(rl.sessionKey, msg.sender)
     ;
}
```

Impact

- Attackers can steal any active session key by re-registering it to their wallet, bypassing all access controls.
- Since `lockedTokens` are tracked by session key (not wallet), hijacked sessions can claim tokens originally locked by the legitimate owner.
- Legitimate owners permanently lose access to their session keys after takeover, as there's no recovery mechanism.

Recommendation

Consider adding the corresponding validation check:

```
if (sessionKeyToWallet[rl.sessionKey] != address(0)) {
    revert();
}
```

Team Response

Fixed.

[C-03] Wrong Wallet Validation in `disableSessionKey()` When Caller is `SESSION_KEY_DISABLER` Role

Severity

Critical Risk

Description

When a `SESSION_KEY_DISABLER` calls `disableSessionKey()`, the function incorrectly checks `sessionData[_sessionKey][msg.sender].validUntil` (which is always 0 for disablers) rather than the target session's expiration time.

This causes the token claim validation to be effectively skipped for privileged callers, as the condition `0 >= block.timestamp` always evaluates to `false`.

Location of Affected Code

File: src/modules/validators/CredibleAccountModule.sol#L159

```
function disableSessionKey(address _sessionKey) external {
// code
  if (sessionData[_sessionKey][msg.sender].validUntil >= block.timestamp
     && !isSessionClaimed(_sessionKey)) {
      revert CredibleAccountModule_LockedTokensNotClaimed(_sessionKey);
  }
// code
}
```

Impact

- Disable active sessions before their `validUntil` time
- Bypass the `LockedTokensNotClaimed` protection completely
- Sessions with unclaimed locked tokens can be forcibly closed

Recommendation

Consider using the `targetWallet` instead of `msg.sender` in the check, as shown:

```
function disableSessionKey(address _sessionKey) external {
// code
- if (sessionData[_sessionKey][msg.sender].validUntil >= block.timestamp
   && !isSessionClaimed(_sessionKey)) {
+ if (sessionData[_sessionKey][targetWallet].validUntil >= block.
   timestamp && !isSessionClaimed(_sessionKey)) {
      revert CredibleAccountModule_LockedTokensNotClaimed(_sessionKey);
  }
// code
}
```

Team Response

Fixed.

[C-04] Memory vs Storage Mismatch Renders Session State Update Failure in `validateUserOp()`

Severity

Critical Risk

Description

The `validateUserOp()` function incorrectly uses `SessionData memory` when updating the live status, causing all session keys to remain permanently active despite validation attempts:

```
SessionData memory sd = sessionData[sessionKeySigner][msg.sender]; //
   Loads into MEMORY
sd.live = false; // Modifies memory copy only
// Storage remains UNCHANGED
```

Location of Affected Code

File: src/modules/validators/CredibleAccountModule.sol#L276

Impact

All validated sessions never deactivate, and `Live = true` persists indefinitely in storage, leading to defeating the entire session expiration mechanism.

Recommendation

Consider using `storage` instead of `memory`:

```
function validateUserOp(PackedUserOperation calldata userOp, bytes32
    userOpHash) external override returns (uint256) {
// code
- SessionData memory sd = sessionData[sessionKeySigner][msg.sender];
+ SessionData storage sd = sessionData[sessionKeySigner][msg.sender];
  sd.live = false;
}
```

Team Response

Fixed.

[C-05] In `CredibleAccountModule` the `validateSessionKeyParams()` Function is `public`

Severity

Critical Risk

Description

The function `CredibleAccountModule::validateSessionKeyParams()` will be used in `validateUserOp()` when validating user input. It changed the `TokenData` to claimed so that the smart wallet can unlock the tokens. But since the function is `public`, anyone can call it with any data and consume any user tokens.

Location of Affected Code

File: src/modules/validators/CredibleAccountModule.sol#L200

```
function validateSessionKeyParams(address _sessionKey,
    PackedUserOperation calldata userOp) public returns (bool) {
     if (isSessionClaimed(_sessionKey)) return false;
     bytes calldata callData = userOp.callData;
     if (bytes4(callData[:4]) == IERC7579Account.execute.selector) {
         ModeCode mode = ModeCode.wrap(bytes32(callData[4:36]));
         (CallType calltype,,,) = ModeLib.decode(mode);
         if (calltype == CALLTYPE_SINGLE) {
             return _validateSingleCall(callData, _sessionKey, userOp.
                sender);
         } else if (calltype == CALLTYPE_BATCH) {
             return _validateBatchCall(callData, _sessionKey, userOp.
                sender);
         }
     }
     return false;
}
```

Impact

Anyone can consume Locked tokens in favour of Smart wallets, preventing them from using the session and unlocking tokens.

Recommendation

The `validateSessionKeyParams()` function should be made `internal` instead of `public`.

Team Response

Fixed.

[C-06] Anyone Can Uninstall the Validator Module from Any Wallet

Severity

Critical Risk

Description

When uninstalling the module, there is no authentication made to the `msg.sender`, the sender parameter is extracted from the data, and we deal with it as if it is the actual sender.

This will allow anyone to call `CredibleAccountModule` directly and `onInstall()` the module (validator/Hook) from any wallet.

Location of Affected Code

File: src/modules/validators/CredibleAccountModule.sol#L327

```
function onUninstall(bytes calldata data) external override {
    if (data.length < 32) {
        revert CredibleAccountModule_InvalidOnUnInstallData(msg.sender);
    }
    uint256 moduleType;
    address sender;
    assembly {
        moduleType := calldataload(data.offset)
>>      sender := calldataload(add(data.offset, 32))
    }
    bytes memory uninstallData = data[32:];
    ...
}
```

Impact

Anyone can uninstall a module from any wallet.

Recommendation

It should be ensured that the `sender` is the same as `msg.sender`.

Team Response

Fixed.

[C-07] In `ResourceLockValidator` the `validateUserOp()` Function Is Not Consuming the Signature Proof

Severity

Critical Risk

Description

When calling `validateUserOp()`, it is called by the Modular wallet from the `EntryPoint` in the beginning, then on our wallet, we call `validateUserOp()` in the `ResourceLockValidator` module.

In case of authenticating `userOp` as that in EIP4337, `EntryPoint` has its own Nonce Mechanism, which validates the tx from replay attacks. But in case of validating against `callData` only, by extracting the `ResourceLock`. There is no Nonce mechanism implemented to only use the signature once.

```
struct ResourceLock {
    uint256 chainId;
    address smartWallet;
    address sessionKey;
    uint48 validAfter;
    uint48 validUntil;
    bytes32 bidHash;
    TokenData[] tokenData;
}
```

This will allow anyone to reuse the Proof Signature again, as they can simply increase the Nonce and call `EntryPoint` directly. The `EntryPoint` will not revert as the nonce is increasing, and the validator will only construct the hash from the `callData`, so the attacker can reuse the signature again and again.

Location of Affected Code

File: src/modules/validators/CredibleAccountModule.sol#L265

```
function validateUserOp(PackedUserOperation calldata userOp, bytes32
   userOpHash) external override returns (uint256) {
    bytes calldata signature = userOp.signature;
    address walletOwner = validatorStorage[msg.sender].owner;
// Standard signature length - no proof packing
    if (signature.length == 65) { ... }
// or if signature.length >= 65 (standard signature length + proof
   packing)
    ResourceLock memory rl = _getResourceLock(userOp.callData);
// Nonce validation
>>  bytes memory ecdsaSignature = signature[0:65];
    bytes32 root = bytes32(signature[65:97]); // 32 bytes
    bytes32[] memory proof;
    ...
    if (!MerkleProofLib.verify(proof, root, _buildResourceLockHash(rl)))
       {
        revert RLV_ResourceLockHashNotInProof();
    }
// check proof is signed
>>  if (walletOwner == ECDSA.recover(root, ecdsaSignature)) {
        return SIG_VALIDATION_SUCCESS;
    }
    bytes32 sigRoot = ECDSA.toEthSignedMessageHash(root);
    address recoveredMSigner = ECDSA.recover(sigRoot, ecdsaSignature);
>>  if (walletOwner != recoveredMSigner) return SIG_VALIDATION_FAILED;
    return SIG_VALIDATION_SUCCESS;
}
```

Impact

Attackers can reuse the signature and force the wallet to execute the transaction again and again.

Proof of Concept

- The Modular Wallet owner installed `ResourceLockValidator`
- The wallet owner made a signature as a Merkle tree
- He submitted a Blob transaction, and `EntryPoint` executed the tx
- Attacker took the `callData` and made another call from the `EntryPoint`, increasing the nonce, and putting the same `callData`
- The `EntryPoint` calls `validateUserOp()` on the sender, which is our Modular wallet in this case
- The validation success
- Nonce check in the `EntryPoint` succeeded
- The `EntryPoint` will call `execute()` on the wallet

Recommendation

- There should be a mechanism to prevent re-executing the tx, in case it is built as `ResourceLock`
  .

- There should be a Nonce management in `ResourceLockValidator`
- Or the Signature should be treated as Consumed (stored in mapping and should not be used again)

Team Response

Fixed.

[C-08] In `ResourceLockValidator`, the `validateUserOp()` Function Lacks Sufficient Checks, Allowing Draining of `ModularEtherspotWallet` Balances

Severity

Critical Risk

Summary

This is a collection of issues with different root causes, we grouped them in a single instance for better explanation.

Description

The `validateUserOp()` function in `ResourceLockValidator` will be called by the Modular wallet `ResourceLockValidator`. This function will be called from the `EntryPoint`, and in case of validation success, the `EntryPoint` will call the `callData` on our wallet. Since the validation succeeded, this means the execution of `callData` is authorised.

If `validateUserOp()`, this means we are calling `callData` on our wallet. So the validity of the data should be correct, as any mistake can result in malicious data that ends up taking all the wallet's funds.

The Core problem in the implementation of `ResourceLockValidator` is the way we validate the data in case of MerkleTree.

File: src/modules/validators/CredibleAccountModule.sol#L265

```
function validateUserOp(PackedUserOperation calldata userOp, bytes32
    userOpHash) ... {
// code
// or if signature.length >= 65 (standard signature length + proof
    packing)
>>  ResourceLock memory rl = _getResourceLock(userOp.callData);
// Nonce validation
    bytes memory ecdsaSignature = signature[0:65];
    bytes32 root = bytes32(signature[65:97]); // 32 bytes
    bytes32[] memory proof;
    ...
    if (!MerkleProofLib.verify(proof, root, _buildResourceLockHash(rl)))
        {
        revert RLV_ResourceLockHashNotInProof();
    }
// check proof is signed
>>  if (walletOwner == ECDSA.recover(root, ecdsaSignature)) {
        return SIG_VALIDATION_SUCCESS;
    }
    bytes32 sigRoot = ECDSA.toEthSignedMessageHash(root);
    address recoveredMSigner = ECDSA.recover(sigRoot, ecdsaSignature);
    if (walletOwner != recoveredMSigner) return SIG_VALIDATION_FAILED;
    return SIG_VALIDATION_SUCCESS;
}
```

The recovery of the signer is done by validating the `root` against the `ecdsaSignature`. The issue is that the root validation is made by hashing `ResourceLock` only, without any further checks. So if attackers provide the same signature/proof but with different `userOp.callData` that results in the same `ResourceLock` signed by the signer, the attacker will be able to pass the validation check and force the Modular wallet to execute their transaction.

The attacker will simply listen to the tx mempool and once he sees the tx made by that signed for that `ResourceLock`, the attacker will front-run it, and call `EntryPoint` himself with his custom data (`userOp.callData`) that results in the same `ResourceLock` at the end. Letting him drain the Modular wallet. Here are all the instances that can occur

Unchecking the target and value

Since we are using `EIP7579`, the function we execute is named `execute()` in the case of a single call. This includes the target (the address we are calling) and value (the amount of ETH we send). The attacker can simply crap the callData and pass his address and the amount of ETH of that modular wallet and call the `EntryPoint`. The final `ResourceLock` will be the same as that signed walletOwner, so this will end up transferring all the ETH in the Modular wallet to the attacker

- We should check if the value is `0` and the target is the `CredibleAccountModule`

Unchecking the length of Batch execution

When making a batch call, we are not checking for the length of the `batchExecs`, we take the first one only and validate it.

File: src/modules/validators/ResourceLockValidator.sol#L197

```
function _getResourceLock(bytes calldata _callData) internal view returns
    (ResourceLock memory) {
    if (bytes4(_callData[:4]) == IERC7579Account.execute.selector) {
        (CallType calltype,,,) = ModeLib.decode(ModeCode.wrap(bytes32(
           _callData[4:36])));
        if (calltype == CALLTYPE_SINGLE) { ... } else if (calltype ==
           CALLTYPE_BATCH) {
// NOTE: If batch call then it will should only contain a single
   UserOperation
// so hardcoded values will hold here
>>          Execution[] calldata batchExecs = ExecutionLib.decodeBatch(
   _callData[100:]);
            for (uint256 i; i < batchExecs.length; ++i) {
              if (bytes4(batchExecs[i].callData[:4]) == bytes4(0x495079a0
                 )) {
                    ...
>>                  return ResourceLock({ ... });
                }
                revert RLV_InvalidSelector();
            }
        } else {
            revert RLV_InvalidCallType();
        }
    }
}
```

So, since the validation of the `callData` is done using the first `execution`, the attacker can take the signed execution from the wallet owner, and make an array including other malicious executions to drain the `ModularEtherspotWallet`, like transferring ERC20 tokens to himself

- We should check that the length is exactly `1`

Unchecking the selector of the `execTarget` to match `enableSessionKey()` in Single execution

`ResourceLockValidator` is intended to be used to enable sessions on `CredibleAccountModule`. The way of encoding and decoding data makes it only work for this. Even the Batch execution enforces the selector to be `0x495079a0`, which is the selector of the `enableSessionKey(bytes)` function.

But in Single execution mode, this selector is not checked. So this will allow consuming the signature in a different selector, putting random data like calling `CredibleAccountModule::onUninstall()` and forcing the uninstallation of the module.

- We should check that the selector of the function we are going to execute is `0x495079a0` same as in Batch execution.

Impact

Draining of the Modular Wallets

Recommendation

Each Sub-issue mitigation is written on it. But for better security. The signature verification should be made to the overall `userOp`, not just the `ResourceLock`, as the caller will be the `EntryPoint` that has access to call `execute()` on the wallet. and `userOp` is the structure that is hashed by `EntryPoint` that includes `chainId`, `nonce` and other parameters that ensure no such signature consumption can be made to drain the wallets.

Team Response

Fixed.

[H-01] No Check for `userOp()` and `userOpHash()` Mismatch Nor the Validity of the Sender

Severity

High Risk

Description

The `CredibleAccountModule` does not hash the `userOp` and uses the hash to know the wallet that authorized the action. It uses `userOpHash` to derive the address that signed for that hash, and uses the data `userOp` to modify it.

File: src/modules/validators/CredibleAccountModule.sol#L272-L273

```
function validateUserOp(PackedUserOperation calldata userOp, bytes32
    userOpHash) ... {
     if (userOp.signature.length < 65) return VALIDATION_FAILED;
     bytes memory sig = _digestSignature(userOp.signature);
>>   address sessionKeySigner = ECDSA.recover(ECDSA.toEthSignedMessageHash
    (userOpHash), sig);
>>   if (!validateSessionKeyParams(sessionKeySigner, userOp)) {
         return VALIDATION_FAILED;
     }
// code
}
```

As we can see, the `sessionKeySigner` is derived from `userOpHash` and the actual input is `userOp`. There is no check that this `userOpHash` is the hashing of that `userOp`, and there is no check that the `userOp.sender` is the actual `msg.sender`. So anyone can do the following.

- Make an attacking contract and install the module
- Listen to the transactions made by the `sessionKeySigner`
- Use the valid hash and provide different data in `userOp` (like changing the recipient of tokens)
- Consume the signature in his favour and prevent consuming it again.
- The real Modular wallet will not be able to execute this tx as it will revert from the `EntryPoint` execution in the validating process.

Impact

Malicious users can set up modules and consume other Valid ModularAccounts' hash. preventing them from consuming the sessionKey signature and releasing the tokens.

Proof of Concept

1. Add the following function in `test/modules/CredibleAccountModule/concrete/CredibleAccountModule.t.sol`

```
function test_validateUserOp_sheildify_sender_not_checked() public
    withRequiredModules {
// Enable session key
     _enableSessionKey(address(scw));
// Claim all tokens by solver
     bytes memory usdcData = _createTokenTransferExecution(solver.pub,
        amounts[0]);
     bytes memory daiData = _createTokenTransferExecution(solver.pub,
        amounts[1]);
     bytes memory usdtData = _createTokenTransferExecution(solver.pub,
        amounts[2]);
     Execution[] memory batch = new Execution[](3);
     batch[0] = Execution({target: address(usdc), value: 0, callData:
        usdcData});
     batch[1] = Execution({target: address(dai), value: 0, callData:
        daiData});
     batch[2] = Execution({target: address(usdt), value: 0, callData:
        usdtData});
     bytes memory opCalldata =
         abi.encodeCall(IERC7579Account.execute, (ModeLib.
            encodeSimpleBatch(), ExecutionLib.encodeBatch(batch)));
     (PackedUserOperation memory op, bytes32 hash) =
         _createUserOpWithSignature(sessionKey, address(scw), address(cam)
            , opCalldata);
```

```
/**
the `sender` will be the SCW that is getting called by the EntryPoint.
If an attacker called `CredibleAccountModule` directly putting `sender`
   as
the real `scw` the `msg.sender` will be the attacker address
**/
    address attacker = makeAddr("attacker");
    vm.stopPrank();
    vm.prank(attacker);
    cam.validateUserOp(op, hash);

// Session is Claimed before the real SCW execute it
    assertTrue(cam.isSessionClaimed(sessionKey.pub));

// This function will revert
    vm.expectRevert();
    vm.startPrank(address(scw));
    _executeUserOp(op);
}
```

Then run the following command:

```
forge test --mt test_validateUserOp_sheildify_sender_not_checked
```

The PoC shows how the session is consumed by an attacker, preventing the Real SCW from consuming it.

Recommendation

It should be ensured that `op.sender` equals `msg.sender`. And an optional thing to do to increase the security is to reconstruct the hash from the `userOp` itself. According to the `EIP-4337` hashing method. Instead of taking he hash and dealing with it as it is the correct hashing of `userOp`.

Team Response

Fixed.

[H-02] In `CredibleAccountModule` the `validateUserOp()` Function Is Not Authenticating the Sender

Severity

High Risk

Description

The `ModularEtherSpotWallets` will activate the `CredibleAccountModule`. By either allowing it as a validator or as a Hook. When working as a validator, it will be used to validate the transaction from that Smart Account by calling `validateUserOp()`. But since the function is not

checking whether the sender is activating the module or not. This will result in anyone calling `CredibleAccountModule::validateUserOp()` and changing the state data.

In `ModularEtherspotWallet::validateUserOp()`, the validator we call should be a validator implemented by that account.

File: src/modules/validators/CredibleAccountModule.sol#L265

```
function validateUserOp(PackedUserOperation calldata userOp, bytes32
    userOpHash) external override returns (uint256) {
// code

// check if validator is enabled. If not, terminate the validation phase.
    if (!_isValidatorInstalled(validator)) return VALIDATION_FAILED;

// bubble up the return value of the validator module
    validSignature = IValidator(validator).validateUserOp(
        userOp,
        userOpHash
    );
}
```

In order for the validator to be installed, the wallet should call `installModule`, putting that validator as one of its validators, and installing a validator we call `onInstall()` on that validator. When calling `onInstall()`, we activate the sender's `Modular wallet` to be an active wallet to use that module.

File: src/modules/validators/CredibleAccountModule.sol#L309

```
function onInstall(bytes calldata data) external override {
    ...
    if (moduleType == MODULE_TYPE_VALIDATOR) {
        ...
>>      moduleInitialized[msg.sender].validatorInitialized = true;
        emit CredibleAccountModule_ModuleInstalled(msg.sender);
    } else if (moduleType == MODULE_TYPE_HOOK) {
        ...
    } else { ... }
}
```

Since there is no check for the caller of `CredibleAccountModule::validateUserOp()`, any address can call it. this includes external addresses, or Modular wallets that implement it as a Hook not a validator, etc. resulting in changing the Module state for wallets that do not take the action throught the EIP4337 `EntryPoint`, making all tokens locked as in validating we increase the claimed tokens, and change other variables, that will lead to break of the module handlign of all Modular wallets use it.

Location of Affected Code

File: src/modules/validators/CredibleAccountModule.sol#L265

```
function validateUserOp(PackedUserOperation calldata userOp, bytes32
   userOpHash) external override returns (uint256) {
    if (userOp.signature.length < 65) return VALIDATION_FAILED;
    bytes memory sig = _digestSignature(userOp.signature);
    address sessionKeySigner = ECDSA.recover(ECDSA.toEthSignedMessageHash
       (userOpHash), sig);
    if (!validateSessionKeyParams(sessionKeySigner, userOp)) {
        return VALIDATION_FAILED;
    }
    SessionData memory sd = sessionData[sessionKeySigner][msg.sender];
    sd.live = false;
    return _packValidationData(false, sd.validUntil, sd.validAfter);
}
```

Impact

- Unauthorised actions made affecting Modular smart wallets that are not made by the Modular wallet itself
- Lock of funds as it will be made as claimed when validating

Recommendation

It should be checked that the `sender` is a Modular wallet that has already installed the module.

```
function validateUserOp(PackedUserOperation calldata userOp, bytes32
   userOpHash)
    external
    override
    returns (uint256)
{
+   if (!moduleInitialized[msg.sender].validatorInitialized) {
+       revert CredibleAccountModule_ModuleNotInstalled(msg.sender);
+   }
// code
}
```

Team Response

Fixed.

[H-03] The `pre/postCheck()` Is Not Authorizing the `wallet` Parameter with `msg.sender`

Severity

High Risk

Description

When using `CredibleAccountModule` as a Hook, the wallet will call `preCheck()` before executing and `postCheck()`. They extract the wallet parameter from the data `sender` and accumulate the tokens for all session keys owned by that wallet

```
function preCheck(address msgSender, uint256 msgValue, bytes calldata
    msgData) external override returns (bytes memory hookData) {
     (address sender,) = abi.decode(msgData, (address, bytes));
     return abi.encode(sender, _cumulativeLockedForWallet(sender));
}
```

The `sender` here is the wallet address; there is no check whether the caller `Modular wallet` is the actual `sender` that we use to cumulatively lock tokens from it. This allows any wallet to accumulate tokens from other wallets, and change their state, breaking the authorization mechanism of session Keys wallet ownership.

Location of Affected Code

File: src/modules/validators/CredibleAccountModule.sol#L542

Impact

Any wallet installed with the Hook module can impersonate using it on another wallet, changing `sesionKey` data

Recommendation

We should check that the `sender` parameter in both `pre/postCheck()` is the `msg.sender`.

Team Response

Fixed.

[H-04] `ModularEtherspotWallet` Can Get DoS'ed from Uninstalling Modules

Severity

High Risk

Description

When installing the `CredibleAccountModule` module, it can be installed as a Validator or as a Hook. To install it as a Validator, the wallet should implement `hookMultiPlexer` as the Hook. And when installing it as a Hook, there are no restrictions.

```
function onInstall(bytes calldata data) external override {
// code
    if (moduleType == MODULE_TYPE_VALIDATOR) {
>>      if (IHookLens(msg.sender).getActiveHook() != address(
   hookMultiPlexer)) {
            revert CredibleAccountModule_HookMultiplexerIsNotInstalled();
        }
        if (!hookMultiPlexer.hasHook(msg.sender, address(this), HookType.
           GLOBAL)) {
            revert CredibleAccountModule_NotAddedToHookMultiplexer();
        }
        moduleInitialized[msg.sender].validatorInitialized = true;
        emit CredibleAccountModule_ModuleInstalled(msg.sender);
    } else if (moduleType == MODULE_TYPE_HOOK) {
>>      moduleInitialized[msg.sender].hookInitialized = true;
    } else {
        revert CredibleAccountModule_InvalidModuleType();
    }
}
```

When uninstalling the module, we do the same things. In order to uninstall it from a validator, the Wallet should implement `hookMultiPlexer` as the Hook. And when uninstalling it from a Hook. It should not be installed as a validator.

```
function onUninstall(bytes calldata data) external override {
// code
    if (moduleType == MODULE_TYPE_VALIDATOR) {
>>      if (IHookLens(sender).getActiveHook() != address(hookMultiPlexer)
   ) {
            revert CredibleAccountModule_HookMultiplexerIsNotInstalled();
        }
        if (!hookMultiPlexer.hasHook(sender, address(this), HookType.
           GLOBAL)) {
            revert CredibleAccountModule_NotAddedToHookMultiplexer();
        }
// code
    } else if (moduleType == MODULE_TYPE_HOOK) {
>>      if (moduleInitialized[sender].validatorInitialized == true) {
            revert CredibleAccountModule_ValidatorExists();
        }
        moduleInitialized[sender].hookInitialized = false;
    } else {
        revert CredibleAccountModule_InvalidModuleType();
    }
}
```

So here is the situation: - To install as Validator/The Hook should be `hookMultiPlexer` - To install as a Hook/nothing prevents you - To uninstall, as Validator / The Hook should be `hookMultiPlexer` - To uninstall as Hook/it should not exist as a validator

This case can put the wallet in a complete DoS state by removing the Hook and validator. Here is the

scenario.

- Etherspot wallet installed `hookMultiPlexer` as Hook
- Etherspot wallet installed `CredibleAccountModule` as validator (succeeded as `hookMultiPlexer` is Hook)
- Etherspot wallet uninstalled `hookMultiPlexer` from Hooks
- Etherspot wallet installed `CredibleAccountModule` as a hook (succeeded as no check when setting it as a hook).

Now, if the wallet in this case, it will reach the DoS state as follows: it will activate `CredibleAccountModule` as both Hook and validator.

- In order to uninstall the validator, `hookMultiPlexer` should be the Hook, and the function should revert.
- In order to uninstall the Hook, it should not be activated as a validator.

The wallet is unable to uninstall the modules, resulting in a persistent, non-recoverable state.

Location of Affected Code

File: src/modules/validators/CredibleAccountModule.sol

Impact

Permanent DoS of `unInstall()` the `CredibleAccountModule` Module from being a validator or Hook.

Recommendation

We should prevent installing the module as a Hook if it is already a validator in the `onInstall()` function.

Team Response

Fixed.

[M-01] The `_removeSessionKey()` Will Never Remove the Unbounded Session Key Array Due to Gas-Based DoS

Severity

Medium Risk

Description

Attacker creates as many session keys for a target wallet as there is no upper limit in `enableSessionKey()` . The attacker locks funds in the last created session key.
Now, when someone(`SESSION_KEY_DISABLER`) tries to disable the session key by calling the `disableSessionKey()` and it calls the `_removeSessionKey()` :

```
// Must iterate through ALL keys to find the target
for (uint256 i; i < keys.length; ++i) {
    if (keys[i] == _sessionKey) {  // Gas runs out here
        keys[i] = keys[keys.length - 1];
        keys.pop();
        break;
    }
}
```

This transaction reverts due to the block gas limit, resulting in the attacker's session key being permanently undecreasable.

Location of Affected Code

File: src/modules/validators/CredibleAccountModule.sol#L387

```
function _removeSessionKey(address _sessionKey, address _wallet) internal
    {
    delete sessionData[_sessionKey][_wallet];
    delete lockedTokens[_sessionKey];
    delete sessionKeyToWallet[_sessionKey];
    address[] storage keys = walletSessionKeys[_wallet];
    for (uint256 i; i < keys.length; ++i) {
        if (keys[i] == _sessionKey) {
            keys[i] = keys[keys.length - 1];
            keys.pop();
            break;
        }
    }
}
```

Impact

- A malicious session cannot be removed
- The `batchDisableSessionKeys()` will be always revert if `SESSION_KEY_DISABLER` mention session of attacker.

Recommendation

Add `MAX_SESSION_KEYS` per `msg.sender`:

```
function enableSessionKey(bytes calldata _resourceLock) external {
// code
  require(
      walletSessionKeys[msg.sender].length < MAX_SESSION_KEYS,
      "Max session keys reached"
  );
// code
}
```

Team Response

Fixed.

[M-02] The `pre/postCheck()` Is Not Checking the Hook Initialization of the Sender

Severity

Medium Risk

Description

In order for the Smart Wallets to use `CredibleAccountModule` as a Hook, they call `onInstall()` and install it as a Hook.

```
function onInstall(bytes calldata data) external override {
// code
    if (moduleType == MODULE_TYPE_VALIDATOR) { ... }
    else if (moduleType == MODULE_TYPE_HOOK) {
>>      moduleInitialized[msg.sender].hookInitialized = true;
    } else {
        revert CredibleAccountModule_InvalidModuleType();
    }
}
```

When executing the tx on `ModularEtherspotWallet`, we call it with the `withHook()` modifier. and since this hook can be installed. The wallet will call `preCheck()` before execution and call `postCheck()` after execution.

```
modifier withHook() {
    address hook = _getHook();
    if (hook == address(0)) {
        _;
    } else {
        bytes memory hookData = IHook(hook).preCheck(
            msg.sender,
            msg.value,
            msg.data
        );
        _;
        IHook(hook).postCheck(hookData);
    }
}
// -----

function execute( ... ) external payable onlyEntryPointOrSelf withHook {
   ... }
```

But in `CredibleAccountModule`, there is no checking whether the `msg.sender` is initialising that Hook or not:

```
function preCheck(address msgSender, uint256 msgValue, bytes calldata
    msgData) external override returns (bytes memory hookData) {
     (address sender,) = abi.decode(msgData, (address, bytes));
     return abi.encode(sender, _cumulativeLockedForWallet(sender));
}

// -----

function postCheck(bytes calldata hookData) external {
// code
}
```

This will make any wallet either a Modular wallet or a Smart wallet, call these functions, allowing them to change the `liveness` of the session keys of other users, breaking the access control of the Locking token mechanism.

Location of Affected Code

File: src/modules/validators/CredibleAccountModule.sol

Impact

Unauthorised access by anyone to session keys allowing them to change the lifespan.

Recommendation

We should check that the caller is a Smart wallet that initialised the Module as Hook.

Team Response

Fixed.

[L-01] Redundant Validation of `validUntil == 0` in the `enableSessionKey()` Function

Severity

Low Risk

Description

The condition `if (rl.validUntil == 0 || rl.validUntil <= rl.validAfter)` contains redundant validation. Since:

- `rl.validAfter == 0` is already checked separately
- If `validUntil == 0`, it would automatically fail `validUntil <= validAfter` (since validAfter must be > 0)

This makes the `validUntil == 0` check unnecessary and potentially confusing.

Location of Affected Code

File: src/modules/validators/CredibleAccountModule.sol#L133

```
function enableSessionKey(bytes calldata _resourceLock) external {
// code
  if (rl.validUntil == 0 || rl.validUntil <= rl.validAfter) {
      revert CredibleAccountModule_InvalidValidUntil(rl.validUntil);
  }
// code
}
```

Recommendation

Consider adjusting the check as follows:

```
if (rl.validUntil <= rl.validAfter || rl.validUntil < block.timestamp) {
      revert CredibleAccountModule_InvalidValidUntil(rl.validUntil);
}
```

Team Response

Fixed.

[L-02] Missing `chainID` Validation and `smartWallet` Validation in Session Key Activation

Severity

Low Risk

Description

The `enableSessionKey()` function processes `ResourceLock` data but fails to validate `chainId` & `smartWallet` fields:

- `chainId` – Not checked against the current blockchain, allowing cross-chain replay (though with limited impact).
- `smartWallet` – Completely ignored, defaulting to `msg.sender` for session binding (unlikely to be exploited in current design).

While these do not immediately threaten funds, they violate best practices for session management.

Location of Affected Code

File: src/modules/validators/CredibleAccountModule.sol#L122

Recommendation

Consider adding a `chainID` and `smartWallet` validation:

- `chainID`

```
    if (rl.chainId != 0 && rl.chainId != block.chainid) {
      revert();
    }
```

- `smartWallet`

```
    address targetWallet = rl.smartWallet != address(0) ? rl.
       smartWallet : msg.sender;
```

Team Response

Fixed.

[L-03] Redundant Selector Validation in `_validateSingleCall()`

Severity

Low Risk

Description

The function `_validateSingleCall()` performs a redundant validation check by calling `_isValidSelector(selector)` after the same check was already executed within the `_digestClaimTx()` function. It creates unnecessary gas overhead and code duplication.

Location of Affected Code

File: src/modules/validators/CredibleAccountModule.sol#L517

```
function _digestClaimTx(bytes calldata _data) internal pure returns (
    bytes4, address, uint256) {
     bytes4 selector = bytes4(_data[0:4]);
     if (!_isValidSelector(selector)) {
         return (bytes4(0), address(0), 0);
     }
     address to = address(bytes20(_data[16:36]));
     uint256 amount = uint256(bytes32(_data[36:68]));
     return (selector, to, amount);
}
```

Recommendation

Consider removing `_isValidSelector(selector)` in `_validateSingleCall()`:

```
function _validateSingleCall(bytes calldata _callData, address
    _sessionKey, address _wallet) internal returns (bool) {
     (address target,, bytes calldata execData) = ExecutionLib.
        decodeSingle(_callData[EXEC_OFFSET:]);
     (bytes4 selector,, uint256 amount) = _digestClaimTx(execData);
// Remove redundant check since _digestClaimTx already validated selector
     return _validateTokenData(_sessionKey, _wallet, amount, target);
}
```

Team Response

Fixed.

[L-04] Redundant Event Emission in Role Management If Role Is Already Given

Severity

Low Risk

Description

The contract's role management system currently emits events unnecessarily when there is no actual state change occurring. Specifically, when: - `_grantRole()` : Granting a role to an account that already has the role - `_revokeRole()` : Revoking a role from an account that doesn't have the role

In these cases, while the parent functions properly return `false` to indicate no state change occurred, the contract still emits the custom `SessionKeyDisablerRoleGranted` or `SessionKeyDisablerRoleRev` events. This creates waste gas.

Location of Affected Code

File: src/modules/validators/CredibleAccountModule.sol#L673

```
function _grantRole(bytes32 role, address account) internal virtual
    override returns (bool) {
     bool result = super._grantRole(role, account);
     if (role == SESSION_KEY_DISABLER) {
         emit SessionKeyDisablerRoleGranted(account, msg.sender);
     }
     return result;
}
```

File: src/modules/validators/CredibleAccountModule.sol#L687

```
function _revokeRole(bytes32 role, address account) internal virtual
    override returns (bool) {
     bool result = super._revokeRole(role, account);
     if (role == SESSION_KEY_DISABLER) {
         emit SessionKeyDisablerRoleRevoked(account, msg.sender);
     }
     return result;
}
```

Recommendation

Consider adjusting the checks in `_grantRole()` and `_revokeRole()` functions:

- `_grantRole()`

```
  if (result && role == SESSION_KEY_DISABLER) {
    emit SessionKeyDisablerRoleGranted(account, msg.sender);
  }
```

- `_revokeRole()`

```
  if (result && role == SESSION_KEY_DISABLER) {
    emit SessionKeyDisablerRoleRevoked(account, msg.sender);
  }
```

Team Response

Fixed.

[L-05] Miner Delay of Accepting the Tx Can Revert Due To `>=` Comparison

Severity

Low Risk

Description

The `disableSessionKey()` function uses `>=` for timestamp comparison when checking `validUntil`, creating a miner-dependent edge condition. If called when `validUntil == block.timestamp`, miners can intentionally delay transaction inclusion to force a revert when: - The session is exactly expiring (`validUntil == block.timestamp`)
- Tokens remain unclaimed (`isSessionClaimed(_sessionKey) == false`)

Location of Affected Code

File: src/modules/validators/CredibleAccountModule.sol#L159

```
function disableSessionKey(address _sessionKey) external {
// code
  if (sessionData[_sessionKey][msg.sender].validUntil >= block.timestamp
     && !isSessionClaimed(_sessionKey)) {
      revert CredibleAccountModule_LockedTokensNotClaimed(_sessionKey);
  }
// code
}
```

Impact

Temporary Denial-of-Service - Miners can selectively censor transactions during the 30-second window where `validUntil == block.timestamp` - Particularly problematic for time-sensitive operations

Recommendation

Consider adding a time buffer of around 30 seconds so that this will not be possible:

```
if (sessionData[_sessionKey][msg.sender].validUntil >= block.timestamp +
   30 sec) {
  revert CredibleAccountModule_LockedTokensNotClaimed(_sessionKey);
}
```

Team Response

Fixed.

[L-06] Missing Timestamp Validation Allows Backdated Session Keys

Severity

Low Risk

Description

The `enableSessionKey()` function fails to validate that `validUntil` is set to a future timestamp (`> block.timestamp`). This could create already-expired sessions by setting `validUntil` to a past or current block timestamp.

Location of Affected Code

File: src/modules/validators/CredibleAccountModule.sol#L133

```
function enableSessionKey(bytes calldata _resourceLock) external {
// code
  if (rl.validUntil == 0 || rl.validUntil <= rl.validAfter) {
      revert CredibleAccountModule_InvalidValidUntil(rl.validUntil);
  }
// code
}
```

Impact

The user can create sessions with `validUntil` set to a past timestamp. While these sessions appear in `walletSessionKeys`, they're immediately unusable since `_isSessionKeyExpired()` marks them inactive.

Recommendation

Consider adding the following validation check:

```
 if (rl.validUntil == 0 || rl.validUntil <= rl.validAfter || rl.
    validUntil <= block.timestamp ) {
      revert CredibleAccountModule_InvalidValidUntil(rl.validUntil);
}
```

Team Response

Fixed.

[L-07] Unbounded Tokens Addition When Enabling SessionKeys

Severity

Low Risk

Description

When locking tokens in a single key, there is a maximum number of tokens set when enabling the key.

```
function enableSessionKey(bytes calldata _resourceLock) external {
// code
    for (uint256 i; i < rl.tokenData.length; ++i) {
>>    lockedTokens[rl.sessionKey].push(
          LockedToken({token: rl.tokenData[i].token, lockedAmount: rl.
             tokenData[i].amount, claimedAmount: 0})
      );
    }
// code
}
```

The number of tokens that can be added within a single key is not limited. This can lead to DoS issues when validating / claiming those tokens, as we will loop through all of them. till we reach the `token` we need.

Location of Affected Code

File: src/modules/validators/CredibleAccountModule.sol#L139

Impact

Denial of service of the validation process because of OOG when looping through all tokens.

Recommendation

We should have a maximum number of tokens that can be added to a single key.

Team Response

Fixed.

[L-08] There Is No Check for the `walletOwner` Existence in the `validateUserOp()` Function in `ResourceLockValidator`

Severity

Low Risk

Description

When validating the user's transaction, we are not checking whether the caller is actually installing the Module as a validator or not.

```
function validateUserOp(PackedUserOperation calldata userOp, bytes32
   userOpHash)
    external
    override
    returns (uint256)
{
    bytes calldata signature = userOp.signature;
>>  address walletOwner = validatorStorage[msg.sender].owner;
// Standard signature length - no proof packing
    if (signature.length == 65) {
// standard ECDSA recover
>>      if (walletOwner == ECDSA.recover(userOpHash, signature)) {
            return SIG_VALIDATION_SUCCESS;
        }
        bytes32 sigHash = ECDSA.toEthSignedMessageHash(userOpHash);
        address recoveredSigner = ECDSA.recover(sigHash, signature);
        if (walletOwner != recoveredSigner) return SIG_VALIDATION_FAILED;
        return SIG_VALIDATION_SUCCESS;
    }
// code
}
```

If the Module is not installed, the address will be `address(0)`. The problem is that the function will return `SIG_VALIDATION_FAILED` instead of reverting, and this violates the EIP4337 standards, where the function should revert if the error is not a signature mismatch.

https://eips.ethereum.org/EIPS/eip-4337#smart-contract-account-interface > MUST validate that the signature is a valid signature of the `userOpHash`, and SHOULD return `SIG_VALIDATION_FAILED` (1) without reverting on signature mismatch. Any other error MUST revert.

Location of Affected Code

File: src/modules/validators/CredibleAccountModule.sol#L265

Impact

Incompatibility with EIP standards

Recommendation

The execution should be reverted if `walletOwner` is a zero address.

Team Response

Fixed.

[L-09] Missing Input Length Validation in the `onInstall()` Function

Severity

Low Risk

Description

The `onInstall()` function directly slices the last 20 bytes of `_data` to derive the owner's address.

```
address owner = address(bytes20(_data[_data.length - 20:]));
```

However, it does not validate whether `_data.length >= 20` before slicing. If the `_data` provided is less than 20 bytes long, this will result in an out-of-bounds read.

Location of Affected Code

File: src/modules/validators/CredibleAccountModule.sol#L287

```
function onInstall(bytes calldata _data) external override {
    address owner = address(bytes20(_data[_data.length - 20:]));
    if (validatorStorage[msg.sender].enabled) {
        revert RLV_AlreadyInstalled(msg.sender, validatorStorage[msg.
            sender].owner);
    }
    validatorStorage[msg.sender].owner = owner;
    validatorStorage[msg.sender].enabled = true;
    emit RLV_ValidatorEnabled(msg.sender, owner);
}
```

Recommendation

Consider adding the missing input data length check:

```
if (_data.length < 20) {
    revert();
}
```

Team Response

Fixed.

shieldify

Thank you!