our shielding · Your smart contracts, our shielding · Your smart c

shieldify

Etherspot

Credible Account Module

Migration Review

SECURITY REVIEW

Date: 11 July 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 (Migration Review)

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 mitigation review lasted 5 days with a total of 40 hours dedicated to the audit by one senior researcher from the Shieldify team.

Overall, the code is well-written. The audit report contributed by identifying one Critical, one High, two Medium and four Low severity findings, mainly related to session key mismanagement, access control flaws, execution order logic, and minor event or validation issues.

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 - Mitigation Review**
Repository	etherspot-modular-accounts
Type of Project	Account Abstraction, ERC-7579, EIP-712
Audit Timeline	5 days
Review Commit Hash	cb0645c3e7ca2a5aa7a6b9ce0263a182fba24fd9
Fixes Review Commit Hash	4a48c318fa6df7e7be0f24fd8c89b6fad828441d

5.2 Scope

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

File	**nSLOC**
src/modules/validators/CredibleAccountModule.sol	481
src/modules/validators/ResourceLockValidator.sol	251
Total	**732**

6. Findings Summary

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

- **Critical** and **High** issues: **2**
- **Medium** issues: **2**
- **Low** issues: **4**

ID	Title	Severity	Status
[C-01]	Consumers Can Double the `lockedAmount` In the Session, Draining the SCW and Preventing Claiming of the Session	Critical	Fixed
[H-01]	Session Key Can Be Consumed by Unauthorized SCW	High	Fixed
[M-01]	Attackers Can Front-Run SCW Preventing Them from Enabling the Session Keys	Medium	Fixed
[M-02]	The `postCheck()` Function in `HookMultiPlexer` Executes in Ascending Order Instead of Descending Order	Medium	Fixed
[L-01]	Owner of the Modular Account in `ResourceLockValidator` Cannot Be Smart Contract Wallet	Low	Fixed
[L-02]	Incorrect Event Emission at `disableSessionKey()` in `CredibleAccountModule`	Low	Fixed
[L-03]	Incorrect `validUntil` Check in `enableSessionKey()` in `CredibleAccountModule`	Low	Fixed
[L-04]	The `DISABLE_SESSION_KEY_TIME_BUFFER` Is Not Used in `batchDisableSessionKeys()`	Low	Fixed

7. Findings

[C-01] Consumers Can Double the `lockedAmount` In the Session, Draining the SCW and Preventing Claiming of the Session

Severity

Critical Risk

Description

When validating the execution of the tx that consumed the locked tokens, the execution can either be a `singleCall()` or `batchCall()`. While validating the batch, we are iterating over all transactions and validating the locked token consumption.

CredibleAccountModule.sol#L450-L461

```
function _validateBatchCall( ... ) ... {
    Execution[] calldata execs = ExecutionLib.decodeBatch(_callData[
       EXEC_OFFSET:]);
    for (uint256 i; i < execs.length; ++i) {
        (bytes4 selector,, uint256 amount) = _digestClaimTx(execs[i].
           callData);
>>      if (!_isValidSelector(selector) || !_validateTokenData(
   _sessionKey, _wallet, amount, execs[i].target)) {
            return false;
        }
    }
    return true;
}
```

We call `_validateTokenData()` in a loop, as the token contract `execs[i].target` . And in `_validateTokenData()` , once the token exists in the `sessionKey` , we increase the `claimedAmount` of it.

CredibleAccountModule.sol#L480

```
function _validateTokenData( ... ) ... {
    LockedToken[] storage tokens = lockedTokens[_sessionKey];
    for (uint256 i; i < tokens.length; ++i) {
        if (tokens[i].token == _token) {
>>          if (_walletTokenBalance(_wallet, _token) >= _amount &&
   _amount == tokens[i].lockedAmount) {
                tokens[i].claimedAmount += _amount;
                return true;
            }
        }
    }
    return false;
}
```

The problem is that we increase the `claimedAmount` if the token matches, and the amount is the same as the `lockedAmount` . Since there is no check to see if the tokens are claimed or not. When executing a batch, users can double/triple/... consume the `lockedAmount` in a Batch execution. As in `_validateTokenData()` , we are not checking if the `claimedAmount` is consumed or not.

The same as in Single execution, if the session consists of 5 tokens, and only one is consumed, the one that is consumed can be consumed again and again.

This will not only allow double unlocking of tokens, but also it will prevent the claiming of the session, where `isSessionClaimed()` checks that `lockedAmount` should equal `claimedAmount` .

CredibleAccountModule.sol#L272

```
function isSessionClaimed(address _sessionKey) public view returns (bool)
    {
    LockedToken[] memory tokens = lockedTokens[_sessionKey];
    for (uint256 i; i < tokens.length; ++i) {
        if (tokens[i].lockedAmount != tokens[i].claimedAmount) return
           false;
    }
    return true;
}
```

Location of Affected Code

File: src/modules/validators/CredibleAccountModule.sol#L480

Impact

- Consuming the `lockedAmount` more than once.
- Preventing `sessionClaiming`, as claimed, will be greater than the `lockedAmount`, making the function always return an unclaimed session.
- Releasing of tokens that should not be released.

Recommendation

We should check that the `claimedAmount` is zero in `_validateTokenData()` so that we are sure the tokens are consumed only on:

```
if (
    _walletTokenBalance(_wallet, _token) >= _amount &&
    _amount == tokens[i].lockedAmount &&
    tokens[i].claimedAmount == 0
) {
    tokens[i].claimedAmount += _amount;
    return true;
}
```

Team Response

Fixed.

[H-01] Session Key Can Be Consumed by Unauthorized SCW

Severity

High Risk

Description

When validating the session to be consumed by calling `CredibleAccountModule::validateUserOp()`. We are enforcing the sender to be a wallet that installed the module and to be `userOp.sender` too.

The issue is that there is no check that `userOpHash` is the actual constructed hash from the `EntryPoint` contract when hashing `PackedUserOperation`.

`sessionKeySigner`, the session key belongs to the actual `msg.sender` or `op.sender`.

CredibleAccountModule.sol#L295-L299

```
function validateUserOp( ... ) ... {
    // code
>>  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];
    return _packValidationData(false, sd.validUntil, sd.validAfter);
}
```

The `sessionKeySigner` is the address that signed `userOp.signature`. After this, we go and validate it.

In validation, there is no check that `sessionKeySigner` is owned by `userOp.sender`. We just check that the **`userOp.sender`** has enough balance to pay in `_validateTokenData()`.

CredibleAccountModule.sol#L473-L487

```
function _validateTokenData(address _sessionKey, address _wallet, uint256
    _amount, address _token) ... {
    LockedToken[] storage tokens = lockedTokens[_sessionKey];
    for (uint256 i; i < tokens.length; ++i) {
        if (tokens[i].token == _token) {
>>          if (_walletTokenBalance(_wallet, _token) >= _amount &&
   _amount == tokens[i].lockedAmount) {
                tokens[i].claimedAmount += _amount;
                return true;
            }
        }
    }
    return false;
}
```

At the end of the `tx` we grad the `SessionData`.

```
function validateUserOp( ... ) ... {
    // code

    if (!_validateSessionKeyParams(sessionKeySigner, userOp)) {
        return VALIDATION_FAILED;
    }
>>  SessionData memory sd = sessionData[sessionKeySigner][msg.sender];
    return _packValidationData(false, sd.validUntil, sd.validAfter);
}
```

If the `sessionKeySigner` does not belong to the `msg.sender/op.sender`, it will return an empty `SessionData`, making `validUntil` and `validAfter` zero. And these values will be bypassed from the `EntryPoint`.

At the end, the `EntryPoint` will execute the session on the SCW that does not own that key, resulting in consuming the session key in an authorized way by non-owners of that key.

Location of Affected Code

File: src/modules/validators/CredibleAccountModule.sol#L300

Impact

- Consumption of the session by non-owners of the session key.
- Preventing consumption sessions by the real owners.
- Bypassing the `validAfter` / `validUntil` check, where the values that will go to the EntryPoint will be zero instead of the real value, because of reading an empty `SessionData` struct.

Proof of Concept

- SCW1 and SCW2 installed the modules
- SCW2 is a malicious wallet
- SCW1 has `key1` to be consumed
- Key1 signed the tx to unlock tokens
- SCW2 took that signature (either from mempool monitoring, front-running it, etc.)
- SCW2 called `CredibleAccountModule.validateUserOp()` providing the following:
  - userOp.signature: the signature made by `key1`
  - userOp.sender: SCW2 (its address)
  - userOpHash: The constructed hash that will be made from the EntryPoint (can be constructed easily by providing the parameters).
- Now, when recovering, the final `sessionKey` will be `key1`, which belongs to `SCW1` and not `SCW2`
- All validations passed, and SCW2 has enough token balance to pass the token validation.
- The `sessionKey` is consumed by `SCW2`, and it is owned by `SCW1`.
- `SCW1` will not be able to consume that session key.

Recommendation

We should check that the consumer of the session is the owner of the recovered `sessionKeySigner` This check can be done as follows.

```
SessionData memory sd = sessionData[sessionKeySigner][msg.sender];
require(sd.sessionKey == sessionKeySigner);
```

NOTE: It is better to include this check in the `_validateSessionKeyParams()` function, which takes `userOp` as input, and `userOp` is the same as `msg.sender`.

Team Response

Fixed.

[M-01] Attackers Can Front-Run SCW Preventing Them from Enabling the Session Keys

Severity

Medium Risk

Description

When enabling `sessionKeys`, it is accessible to anyone to enable any `sessionKey` they want. And after enabling the `sessionKey`, it can't be reused again.

CredibleAccountModule.sol#L133-L134

```
function enableSessionKey(bytes calldata _resourceLock) external {
    // code
>>  if (sessionKeyToWallet[rl.sessionKey] != address(0)) {
        revert CredibleAccountModule_SessionKeyAlreadyExists(rl.
            sessionKey);
    }
    // code
    walletSessionKeys[msg.sender].push(rl.sessionKey);
>>  sessionKeyToWallet[rl.sessionKey] = msg.sender;
    emit CredibleAccountModule_SessionKeyEnabled(rl.sessionKey, msg.
       sender);
}
```

[language=solidity]

So once `sessionKey` is enabled, no one can enable the same key. Unless the key is removed.

The problem comes from the way the modules are integrated with each other, and how the real `ModularEtherspotWallet` will enable session keys.

In order for `ModularEtherspotWallet` to `enableSessionKeys()`, they use `ResourceLockValidator`, as one of their validators, where it is responsible for setting up an `owner` for the `ModularEtherspotWallet`, and this `owner` will sign a tx to enable new session keys.

In `ResourceLockValidator`, which is a normal validator Module, will execute `ICredibleAccountModule.enableSessionKey()`. So once the validation succeeds, the `EntryPoint` will call execute on the SCW, and it will execute `ICredibleAccountModule.enableSessionKey()`. But since there is no check for the `sender <-> sessionkey` relation, anyone can front-run this execution, and know the `sessionKey` that is to be used by the SCW, call `ICredibleAccountModule.enableSessionKey()` directly with that `sessionKey`, with their random parameters (`lockedAmount`, `validAfter`, `validUntil`). and prevents the SCW from using its `sessionKey`.

Location of Affected Code

File: src/modules/validators/CredibleAccountModule.sol#L133-L134

Impact

Greifying SCWs by attackers, preventing them from enabling their `sessionKeys` for usage.

Proof of Concept

- SCW installed `ResourceLockValidator` as a validator
- The owner of that SCW on `ResourceLockValidator` signed a tx for a `ResourceLock`, using a sessionKey (Key001)
- Attacker listened to the mempool and found that this SCW is going to enable `Key001`.
- Attacker called `ICredibleAccountModule.enableSessionKey()`, and installed that key for himself
- When executing the tx on the SCW, it will revert as the sessionKey is already used
- SCW will be prevented from using `ResourceLockValidator` to enable the session keys, as anyone can frontrun them and prevent them from enabling the `sessionKey` they own

Recommendation

The `ResourceLockValidator` should store an enumerable of `SCW -> sessionKey`. And when calling `ICredibleAccountModule.enableSessionKey()`, we check that the `sessionKey` exists in the `ResourceLockValidator` as one of the keys for that SCW.

Team Response

Fixed.

[M-02] The `postCheck()` Function in `HookMultiPlexer` Executes in Ascending Order Instead of Descending Order

Severity

Medium Risk

Description

The `HookMultiPlexer` is designed to accept adding more than one Hook, the Contract will execute each added Hook `preCheck()` function, then after the execution, it will execute `postCheck()`.

The problem is that the execution of both `preCheck()` and `postCheck()` is done in ascending order.

HookMultiPlexer.sol#L327-L331

```
function postCheck(bytes calldata hookData) external override {
    // create the hooks and contexts array
    HookAndContext[] calldata hooksAndContexts;
    // code
>>  for (uint256 i; i < length; i++) {
        // cache the hook and context
        HookAndContext calldata hookAndContext = hooksAndContexts[i];
        // call postCheck on each hook
        hookAndContext.hook.postCheckSubHook({preCheckContext:
           hookAndContext.context});
    }
}
```

This will result in incorrect Hook executions if there is more than one Hook to be implemented, especially when one Hook depends on the other.

Example

Let's say we have three installed hooks, the order is `H1`, `H2` and `H3`. In normal operation, the following should be done. - `H1` called `preCheck()` function - `H1` called execution block3 - `H2` exists. call `H2::preCheck()` - `H3` exists. call `H3::preCheck()` - Main tx execution is done - Now returning back.. `H3` should be the first to execute the `postCheck()` as it only has a block of the main tx, but as `H2` contains `H3` inside, the block of execution is ended after `H3::preCheck()/postCheck()` functions end.

The normal execution should be as follows:

```
-- H1::pre
---- H2::pre
------ H3::pre
--------tx execution
------ H3::post
---- H2::post
-- H1::post
```

But since in `postCheck()` in the `HookMultiPlexer` we perform ascending order, the execution will be as follows:

```
-- H1::pre
---- H2::pre
------ H3::pre
--------tx execution
------ H1::post
---- H2::post
-- H3::post
```

The execution order will not be correct, which can result in incorrect execution if some hook depends on others.

Here is a reference to the MetaMask delegation framework, and they are using the correct order, by doing the post Hook functionality descendingly.

MetaMask/delegation-framework::DelegationManager.sol#L229-L273

```
for (uint256 batchIndex_; batchIndex_ < batchSize_; ++batchIndex_) {
    if (batchDelegations_[batchIndex_].length == 0) {
        ...
    } else {
        // Execute beforeHooks
>>      for (uint256 delegationsIndex_; delegationsIndex_ <
   batchDelegations_[batchIndex_].length; ++delegationsIndex_) {
            // code
        }

        // Perform execution
        IDeleGatorCore(batchDelegations_[batchIndex_][batchDelegations_[
           batchIndex_].length - 1].delegator)
            .executeFromExecutor(_modes[batchIndex_], _executionCallDatas
               [batchIndex_]);

        // Execute afterHooks
>>      for (uint256 delegationsIndex_ = batchDelegations_[batchIndex_].
   length; delegationsIndex_ > 0; --delegationsIndex_)
        {
            // code
        }
    }
}
```

Location of Affected Code

File: src/modules/hooks/HookMultiPlexer.sol#L327-L331

Impact

The incorrect order of Hooks `postCheck()` function will result in unexpected results, especially if some hooks depend on others.

Recommendation

We should order the `postCheck()` function in descending order.

Team Response

Fixed.

[L-01] Owner of the Modular Account in `ResourceLockValidator` Cannot Be Smart Contract Wallet

Severity

Low Risk

Description

When recovering the signature of the `walletOwner`. We use the ECDSA method for recovery. This prevents modular wallets from setting the owner to a smart contract wallet that uses `eip-1271`.

ResourceLockValidator.sol#L166

```
function validateUserOp(PackedUserOperation calldata userOp, bytes32
    userOpHash) ... {
     // code

     if (!MerkleProofLib.verify(proof, root, _buildResourceLockHash(rl)))
        {
         revert RLV_ResourceLockHashNotInProof();
     }
     if (consumedBidHashes[userOp.sender].contains(rl.bidHash)) {
         revert RLV_BidHashAlreadyConsumed(rl.bidHash);
     }
     // check proof is signed
>>   if (walletOwner == ECDSA.recover(root, ecdsaSignature)) {
         consumedBidHashes[userOp.sender].add(rl.bidHash);
         return SIG_VALIDATION_SUCCESS;
     }
     bytes32 sigRoot = ECDSA.toEthSignedMessageHash(root);
>>   address recoveredMSigner = ECDSA.recover(sigRoot, ecdsaSignature);
     if (walletOwner != recoveredMSigner) return SIG_VALIDATION_FAILED;
     consumedBidHashes[userOp.sender].add(rl.bidHash);
     return SIG_VALIDATION_SUCCESS;
}
```

This prevents setting the `owner` to a smart contract wallet that does not use a private key signing schema.

Location of Affected Code

File: src/modules/validators/ResourceLockValidator.sol#L166

File: src/modules/validators/ResourceLockValidator.sol#L171

Impact

Inability for the Modular Wallet that uses `ResourceLockValidator` to set up the owner for that module with a smart contract wallet.

Recommendation

We can use `SignatureCheckerLib` lib instead of direct ECDSA, which will allow Modular wallets to use any kind of wallet as their owner instead of just EOA type.

Team Response

Fixed.

[L-02] Incorrect Event Emission at `disableSessionKey()` in `CredibleAccountModule`

Severity

Low Risk

Description

Both `SESSION_KEY_DISABLER` and the owner of the `sessionKey` can disable the `sessionKey`. They can call `disableSessionKey()`, and the key will be disabled if it is claimed or has reached the disabling period.

After executing the function, we emit `CredibleAccountModule_SessionKeyDisabled`, which takes two parameters, `sessionKey`, and the wallet.

The problem is that the second parameter of the event is `msg.sender` instead of the `targetWallet`.

CredibleAccountModule.sol#L177

```
function disableSessionKey(address _sessionKey) external {
    address sessionOwner = sessionKeyToWallet[_sessionKey];
    if (!hasRole(SESSION_KEY_DISABLER, msg.sender) && msg.sender !=
       sessionOwner) {
        revert CredibleAccountModule_UnauthorizedDisabler(msg.sender);
    }
>>  address targetWallet = sessionOwner != address(0) ? sessionOwner :
   msg.sender;
    // code
    _removeSessionKey(_sessionKey, targetWallet);
>>  emit CredibleAccountModule_SessionKeyDisabled(_sessionKey, msg.sender
   );
}
```

This will result in emitting the event with the incorrect wallet in case the caller was `SESSION_KEY_DISABLER`.

Location of Affected Code

File: src/modules/validators/CredibleAccountModule.sol#L177

Impact

Incorrect event data emission. This can lead to incorrect data retrievals from the blockchain and hard monitoring.

Recommendation

Consider using `targetWallet` instead of `msg.sender`.

Team Response

Fixed.

[L-03] Incorrect `validUntil` Check in `enableSessionKey()` in `CredibleAccountModule`

Severity

Low Risk

Description

When checking the validity of the `validUntil` parameter of the `ResourceLock`, we enforce the value to be greater than the current `block.timestamp`, where if the value is the same as `block.timestamp`, we revert the tx.

CredibleAccountModule.sol#L139-L141

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

The problem is that it is acceptable that the `validUntil` for `EntryPoint` is the same as the current `block.timestamp`, where `outOfTimeRange` will only occur if the current `block.timestamp` is greater than `validUntil`.

This is the same in this system design, where a session can be consumed normally if `validUntil` equals `block.timestamp` and it only deals to be expires if `validUntil < block.timestamp`

CredibleAccountModule.sol#L524

```
function _isSessionKeyExpired(address _sessionKey, address _wallet)
   internal returns (bool) {
    SessionData storage sd = sessionData[_sessionKey][_wallet];
    if (!sd.live) {
        return true;
>>  } else if (sd.validUntil < block.timestamp && sd.live) {
        ...
    } else {
        return false;
    }
}
```

Location of Affected Code

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

Impact

Valid periods of `validUntil` will be rejected by the Module.

Recommendation

Consider using `<` instead of `<=` so that we accept `validUntil` to be the same as `timestamp`
.

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

Team Response

Fixed.

[L-04] The `DISABLE_SESSION_KEY_TIME_BUFFER` Is Not Used in `batchDisableSessionKeys()`

Severity

Low Risk

Description

When disabling session keys, there is a buffer period amount, which represents an amount of time that we open `disableSessionKey()` for it, even if it is not yet expired.

If the session is not claimed, and there are only a few seconds remaining, smaller than the buffer number, disabling the session key is acceptable.

CredibleAccountModule.sol#L171

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

This mechanism is only implemented in a single disabling `sessionKey`, but in `batchDisableSessionKeys()`, this does not exist, and it only accepts disabling the key if it is expired without the buffer period.

CredibleAccountModule.sol#L189

```
function batchDisableSessionKeys(address[] calldata _sessionKeys)
   external onlyRole(SESSION_KEY_DISABLER) {
    for (uint256 i; i < _sessionKeys.length; i++) {
        // code
        // Check if session has expired or all tokens are claimed
>>      bool isExpired = block.timestamp > sessionData[sessionKey][
   targetWallet].validUntil;
        bool allTokensClaimed = isSessionClaimed(sessionKey);
        // code
    }
}
```

This will prevent disabling the sessionKey at the `time_buffer` period before the actual expiration.

Location of Affected Code

File: src/modules/validators/CredibleAccountModule.sol#L189

Impact

Inability to disable the session keys in batches in the `time_buffer` period.

Recommendation

We should subtract the buffer from the `validUntil` when determining the expiration.

```
bool isExpired = block.timestamp > sessionData[sessionKey][targetWallet].
   validUntil - DISABLE_SESSION_KEY_TIME_BUFFER;
```

Team Response

Fixed.

our shielding · Your smart contracts, our shielding · Your smart c

shieldify

Thank you!