SPEARBIT

Stackup Keystore Security Review

Auditors

Gerard Persoon, Lead Security Researcher

Kaden, Lead Security Researcher

Tnch, Security Researcher

Hake, Associate Security Researcher

Report prepared by: Lucas Goiriz

August 18, 2025

Contents

1 About Spearbit

Spearbit is a decentralized network of expert security engineers offering reviews and other security related services to Web3 projects with the goal of creating a stronger ecosystem. Our network has experience on every part of the blockchain technology stack, including but not limited to protocol design, smart contracts and the Solidity compiler. Spearbit brings in untapped security talent by enabling expert freelance auditors seeking flexibility to work on interesting projects together.

Learn more about us at spearbit.com

2 Introduction

Stackup helps businesses manage their crypto assets.

Disclaimer: This security review does not guarantee against a hack. It is a snapshot in time of Stackup Keystore according to the specific commit. Any modifications to the code will require a new security review.

3 Risk classification

Severity level	Impact: High	Impact: Medium	Impact: Low
Likelihood: high	Critical	High	Medium
Likelihood: medium	High	Medium	Low
Likelihood: low	Medium	Low	Low

3.1 Impact

- High - leads to a loss of a significant portion (>10%) of assets in the protocol, or significant harm to a majority of users.
- Medium - global losses <10% or losses to only a subset of users, but still unacceptable.
- Low - losses will be annoying but bearable--applies to things like griefing attacks that can be easily repaired or even gas inefficiencies.

3.2 Likelihood

- High - almost certain to happen, easy to perform, or not easy but highly incentivized
- Medium - only conditionally possible or incentivized, but still relatively likely
- Low - requires stars to align, or little-to-no incentive

3.3 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

4 Executive Summary

Over the course of 2 days in total, Stackup engaged with Spearbit to review the stackup-keystore protocol. In this period of time a total of **40** issues were found.

Summary

Project Name	Stackup
Repository	stackup-keystore
Commit	696c0900
Type of Project	Wallet, Smart Account
Audit Timeline	Jul 27th to Jul 29th

Issues Found

Severity	**Count**	**Fixed**	**Acknowledged**
Critical Risk	0	0	0
High Risk	0	0	0
Medium Risk	3	3	0
Low Risk	7	6	1
Gas Optimizations	4	2	2
Informational	26	21	5
Total	**40**	**32**	**8**

5 Findings

5.1 Medium Risk

5.1.1 What if a wrong `rootHash` is used?

Severity: Medium Risk

Context: Keystore.sol#L17, Keystore.sol#L33

Description: Assume a wrong `rootHash` is created, for example due to a bug in the creation script. Once that is updated via `handleUpdates()` then is no way to do any more updates or validations and the account would be bricked.

Recommendation: Consider creating a recovery mechanism for this situation.

Stackup: Fixed in PR 57.

Spearbit: Fix verified.

5.1.2 Signature from one address can be recycled to maliciously reach threshold

Severity: Medium Risk

Context: UserOpMultiSigVerifier.sol#L51-L52

Description: The `UserOpMultiSigVerifier` contract allows signature recycling when the owners array contains duplicate addresses. This is due to the fact that the contract uses a `seen` array to track which indices have been used, but does not prevent the same signer address from being referenced multiple times if it appears at different positions in the `owners` array. Note that the contract prevents reusing the same index but not the same signer address.

As a consequence, a rogue signer can satisfy the multisig threshold using fewer unique signatures than intended. This vulnerability undermines the security guarantees of the multisig verifier, where a single compromised key could control the entire multisig if the associated account appears multiple times in the `owners` array. This would allow actions to be executed without the expected multi-party authorization.

Proof of Concept:

```
event Signers(Signer[]);
function _createSignersRepeatedSigner(uint8 size) internal returns (Signer[] memory) {

    Signer[] memory signers = new Signer[](size);
    (address addr, uint256 pk) = makeAddrAndKey("Alice");
    for (uint8 i = 0; i < size; i++) {
        signers[i] = Signer({addr: addr, pk: pk});
    }
    emit Signers(signers);
    return signers;
}
function testFuzz_validateDataRepeatedSigner(bool withUserOp,uint8 threshold, uint8 offset, uint8 size)
↪   public {
    _assume(threshold, offset, size);
    Signer[] memory signers = _createSignersRepeated(size);
    bytes32 message = keccak256("Signed by signer");

    bytes memory data = _createData(message, threshold, offset, signers);
    if (withUserOp) {
        PackedUserOperation memory userOp;
        userOp.signature = data;
        data = abi.encode(userOp);
    } else {
        data = abi.encodePacked(verifier.SIGNATURES_ONLY_TAG(), data);
    }
    bytes memory config = _createConfig(threshold, signers);

    uint256 validationData = verifier.validateData(message, data, config);
    assertEq(validationData, SIG_VALIDATION_SUCCESS);
}
```

Recommendation: Modify the validation logic to ensure owner addresses are unique. One possible option would involve tracking used signer addresses instead of just indices. Alternatively, signer's uniqueness could be verified by requiring the signatures to be provided in ascending or descending order, allowing for the verification of duplicates, for example:

```
address lastSigner;
for (uint256 i = 0; i < signatures.length; i++) {
    SignerData memory sd = signatures[i];
    require(owners[sd.index] > lastSigner, "Signatures must be sorted");

    // Note: we need to ensure gas usage is consistent during simulation with dummy signers.
    owners[sd.index] == ECDSA.recover(message, sd.signature) ? valid++ : invalid++;

    lastSigner = owners[sd.index];
}
```

Stackup: Fixed in PR 47.

Spearbit: Fix verified.

5.1.3 Signature replay possible with shared signers

Severity: Medium Risk

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

Description: In `KeystoreAccount.isValidSignature`, we pass the provided `hash` and deconstructed `signature` to the verifier, returning the corresponding value accordingly:

```
function isValidSignature(bytes32 hash, bytes memory signature)
    public
    view
    virtual
    override
    returns (bytes4 magicValue)
{
    (bytes memory proof, bytes memory node, bytes memory data) = abi.decode(signature, (bytes, bytes,
    ↪  bytes));
    ValidateAction memory action =
        ValidateAction({refHash: _refHash, message: hash, proof: proof, node: node, data: data});
    if (IKeystore(_keystore).validate(action) == SIG_VALIDATION_FAILED) {
        return ERC1271_INVALID_VALUE;
    }
    return ERC1271_VALID_VALUE;
}
```

In case the `hash` doesn't specify this contract address (which is common for many applications, e.g. Permit2), and the same signer is used for another account, the signature, once broadcast from one account, could be replayed on the other account unexpectedly.

This may not only occur in the case of shared `UserOpECDSAVerifier` signers, but it also affects multisig signers. For example, if an EOA is a signer on multiple multisigs or a multisig and an individual ECDSA verifier, their signature can be replayed on each of them.

Recommendation: Implement a defensive rehashing scheme as specified by ERC-7739

Stackup: Fixed in PR 48.

Spearbit: Fix verified.

5.2 Low Risk

5.2.1 Function `handleUpdates` could be reentered

Severity: Low Risk

Context: Keystore.sol#L17-L40, Keystore.sol#L94-L96

Description: A malicious verifier could reenter `handleUpdates()` while the `nonce` isn't incremented yet. This way it could potentially allow processing the same `action` again and increase the `nonce` multiple times. Luckily `validateData()` is a view function and thus updates (even when reentered) can't be made, which lowers the potential impact.

Note: a malicious verifier could also just allow every action, which would be worse.

Recommendation: If the `validateData()` would ever be changed to allow updates or an other external call is added that allows updates:

- Consider adding a `nonReentrant` modifier.
- Alternatively consider checking in `_incrementNonce()` the correct `nonce` value is still present.

Stackup: Fixed in PR 52.

Spearbit: Fix verified.

5.2.2 Multisig should have minimum threshold of one

Severity: Low Risk

Context: UserOpMultiSigVerifier.sol#L35

Description: Setting `threshold == 0` shouldn't be allowed as it makes it possible to pass in zero valid signatures and still pass the validation.

Recommendation: Consider requiring `threshold > 0`.

Stackup: Fixed in PR 42.

Spearbit: Fix verified.

5.2.3 `ECDSA.recover()` reverts on failure

Severity: Low Risk

Context: UserOpECDSAVerifier.sol#L22-L38, UserOpMultiSigVerifier.sol#L28, UserOpMultiSigVerifier.sol#L51, UserOpWebAuthnCosignVerifier.sol#L43

Description: `ECDSA.recover()` is used in several verifiers. However this functions reverts if no valid address can be recovered. This way no alternative signature check can be used.

Recommendation: Consider using `ECDSA.tryRecover()`.

Stackup: This is the intended behaviour. i.e. it should revert on an `InvalidSignature()` error if a bad signature is given. This is also how most ECDSA based smart accounts work (e.g. SimpleAccount).

Note that this won't interfere with gas estimations with dummy signers since those signatures are still expected to be structurally valid. If dummy signatures were allowed to be invalid then the gas estimation would be inaccurate anyways.

This expected behaviour has been explicitly outlined in PR 43.

Spearbit: Acknowledged.

5.2.4 `UserOpMultiSigVerifier::validateData()` fails with more than 256 owners

Severity: Low Risk

Context: UserOpMultiSigVerifier.sol#L14-L17, UserOpMultiSigVerifier.sol#L28-L35

Description: `UserOpMultiSigVerifier::validateData()` can fail with more than 256 owners, because `index` is only an `uint8` and the last owners can't be indexed.

Recommendation: Consider checking `owners.length() <= type(uint8).max`. This should also be checked when generating the merkletree.

Stackup: Fixed in PR 44.

Spearbit: Fix verified.

5.2.5 `UserOpMultiSigVerifier::validateData()` fails with more than 256 signatures

Severity: Low Risk

Context: UserOpMultiSigVerifier.sol#L28, UserOpMultiSigVerifier.sol#L44-L53

Description: If `signatures.length` is larger than 256 then potentially `valid` or `invalid` can be increased to a value that no longer fits in an `uint8`.

Recommendation: Consider checking `signatures.length() <= type(uint8).max`.

Stackup: Fixed in PR 45.

Spearbit: Fix verified.

5.2.6 Excluding the `chainId` may cause unexpected effects

Severity: Low Risk

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

Description: In `Keystore.handleUpdates`, we intentionally exclude the `chainId` from the signed `message`:

```
bytes32 message =
    keccak256(abi.encode(action.refHash, action.nextHash, action.account, action.nonce, nodeHash));
```

This is done for improved UX, allowing for accounts to easily have the same verifiers on all chains. However, there are several potential risks to this approach. The first category of risk to consider is the variance in behavior and protocol availability between chains. For example, in case a verifier includes logic which depends on the duration of blocks, the outcome can vary significantly between chains. Similarly, in case a verifier depends on some external contract, e.g. a token, it may not be present on that chain, or there may be another contract deployed to that address, executing malicious logic.

The second category of risk to consider is that the verifiers themselves must be safely deployed such that another contract cannot be deployed to the same address on a different chain. This is possible in case the verifier contract or any parent contract is deployed by a shared factory. One potential case in which this may occur is if the verifier has been deployed by a gnosis safe, whereby an attacker may deploy a safe that they control at the same address on a different chain, allowing them to deploy a different contract to the verifier address, potentially stealing funds from users of that verifier.

Recommendation: Include the current `chainId` in the signed `message`, potentially also including logic to make it optional. Otherwise, at least clearly document this potential risk, explaining how to safely deploy verifiers.

Stackup: Fixed in PR 50.

Spearbit: Fix verified.

5.2.7 Upgradeability pattern inconsistency in `KeystoreAccount` contract

Severity: Low Risk

Context: KeystoreAccount.sol#L15

Description: The `KeystoreAccount` contract implements an inconsistent upgradeability pattern. The `KeystoreAccountFactory` deploys all account instances behind `ERC1967` proxies, which indicates an intention for these contracts to be upgradeable. However, the `KeystoreAccount` implementation does not implement functions to handle upgradeability (for example, it does not inherit from OpenZeppelin's `UUPSUpgradeable` contract), which means the deployed proxies lack the necessary functions to perform upgrades.

It's worth noting that the ERC-4337 explicitly states that "most of ERC-4337 Smart Contract Account will be upgradeable" So, this mismatch creates unnecessary ambiguity that may result in unexpected behaviours or missing intended functionality.

Recommendation: Consider documenting the intended upgradeability behavior clearly in the system specification to prevent future confusion, and implement the missing upgradeability features (with the necessary access controls) if `KeystoreAccount` contracts are intended to be upgradeable.

Stackup: For maximum simplicity, the `KeystoreAccount` does NOT have a built-in path for upgradeability. Fixed in PR 49.

Spearbit: Fix verified.

5.3 Gas Optimization

5.3.1 Cache read directly after write

Severity: Gas Optimization

Context: KeystoreAccount.sol#L46-L57, Keystore.sol#L42-L43, Keystore.sol#L113-L120

Description: In function `_validateSignature()`, if a proof is present, it always stores the `action.node` in the cache via `registerNode()`. And then `Keystore::validate()` retrieves it again with `_validateNode()`.

Recommendation: Consider moving the cache writing logic to the contract `Keystore`. This way retrieval can be skipped if it has just has been stored.

Stackup: This is intended behaviour. The `KeystoreAccount` makes a simple assumption to cache all nodes. However we recognise that this might not be the case for all account implementations that choose to integrate with `Keystore`.

For example, if I have a one off recovery node (such as an multi-sig guardian set) then I don't really need to spend the extra storage cost to cache it. Hence why `registerNode` is not included within the validate path.

Spearbit: Acknowledged.

5.3.2 Array lenght in `for` loop is inefficient

Severity: Gas Optimization

Context: Keystore.sol#L18, UserOpMultiSigVerifier.sol#L47

Description: Several for loops use the array length, which has to be calculated every interation. This is inefficient.

Recommendation: Consider caching the array length in for loops.

Stackup: Fixed in PR 22.

Spearbit: Fix verified.

5.3.3 `verifyCalldata()` can be used

Severity: Gas Optimization

Context: Keystore.sol#L49-L55

Description: Function `registerNode()` uses `MerkleProofLib.verify()`. Because the `proof` is calldata, a more efficient version can be used.

Recommendation: Consider using `verifyCalldata()`.

Note: this is no longer relevant if the suggestion for "Different proof type used in `registerNode()`" is implemented.

Stackup: Fixed in PR 24 (as per comment above).

Spearbit: Fix verified.

5.3.4 Value could already be in the cache

Severity: Gas Optimization

Context: KeystoreAccount.sol#L46-L54, Keystore.sol#L49-L58

Description: In function `registerNode()` the value could already be in the cache, especially because `_validateSignature()` always calls `registerNode()` if a proof is present. It would save some gas to retrieve it the value first and check if it is avaible. At least you can then skip the `MerkleProofLib.verify()`.

Recommendation: Consider checking the node is already present before checking the proof and adding the node.

Stackup: This would only be a gas optimisation for the non ideal case where the wallet doesn't check that a proof is already cached before creating a transaction which triggers `registerNode`. In such a scenario, `registerNode` would still be idempotent (i.e. the same node gets saved again).

Hence why we left out the check and optimised for the ideal case which saves on the unnecessary storage load and conditional check.

Spearbit: Acknowledged.

5.4 Informational

5.4.1 String based error

Severity: Informational

Context: KeystoreAccountFactory.sol#L23-L24, UserOpECDSAVerifier.sol#L14, UserOpMultiSigVerifier.sol#L20, UserOpWebAuthnCosignVerifier.sol#L17, UserOpWebAuthnVerifier.sol#L14

Description: In most situations a custom error is used, however in some places a string based error is used. This isn't consistent.

Recommendation: Consider using custom errors everywhere.

Stackup: Fixed in PR 25.

Spearbit: Fix verified.

5.4.2 Permanent stake and configuration of time

Severity: Informational

Context: KeystoreAccountFactory.sol#L51-L53

Description: Function `addPermanentEntryPointStake()` adds a permanent stake because there is no code to retrieve the stake. It also specifies a `unstakeDelaySec`, however this doesn't provide much value because the stake is permanent.

Recommendation: Consider using a fixed large value for the `unstakeDelaySec`, when calling `entryPoint.addStake()` and remove the parameter from `addPermanentEntryPointStake()`.

Stackup: Fixed in PR 26.

Spearbit: Fix verified.

5.4.3 Public variable starts with _

Severity: Informational

Context: KeystoreAccount.sol#L15-L16

Description: Public variables normally don't start with an _, because that would indicate an internal variable.

Recommendation: Consider doing on of the following:

- Rename `_refHash` to `refHash`.
- Make `_refHash internal` and add a `view` function `refHash()` to retrieve the value.

Stackup: Fixed in PR 27.

Spearbit: Fix verified.

5.4.4 Cache mechanism not documented in specification

Severity: Informational

Context: spec.md#L13, KeystoreAccount.sol#L46-L57

Description: The cache mechanism is not documented in specification.

Recommendation: Consider updating the specification to include the caching patterns:

If cache is not used:

- `Action.proof` is the Merkle tree proof.
- `Action.node` is the entire unhashed node.

If cache is used:

- `Action.proof` is empty.
- `Action.node` is the keccak256 hash of the node.

Stackup: Fixed in PR 28.

Spearbit: Fix verified.

5.4.5 Conversion to bytes not obvious

Severity: Informational

Context: KeystoreAccount.sol#L46, KeystoreAccount.sol#L56, Keystore.sol#L113-L119

Description: Function `_validateSignature()` uses `abi.encode()` to convert from `bytes32` to `bytes`. This works because `bytes32` is exacly one EVM word, however with other datatypes it would not work because then also the length would be encoded and/or the length is increased to a multiple of EVM words.

Note: `_validateNode()` does the reverse operation.

Recommendation: Consider using `bytes.concat()`, which is more obvious and also add a comment:

```diff
- action.node = abi.encode(keccak256(action.node));
+ action.node = bytes.concat(keccak256(action.node)); // convert from bytes32 to bytes
```

Also consider adding a comment in `_validateNode()`.

```
  function _validateNode(/*...*/) // ...
      // ...
-     nodeHash = bytes32(aNode);
+     nodeHash = bytes32(aNode); // convert from bytes to bytes32
      // ...
  }
```

Stackup: Fixed in PR 29.

Spearbit: Fix verified.

5.4.6 Possible reverts in loop of `handleUpdates()`

Severity: Informational

Context: Keystore.sol#L17-L40, Keystore.sol#L85-L92, Keystore.sol#L113-L129

Description: Function `handleUpdates()` tries to continue if an error occurs in `verifier::validateData()`. However reverts can also occur in `_validateAndGetNonce()` and `_validateNode()`, in which case the loop stops and transaction is reverted.

Recommendation: Doublecheck how robust the loop should be and consider handling errors in `_validateAndGetNonce()` and `_validateNode()` without a revert.

Stackup: Acknowledged. We are ok with the possibility of a `handleUpdates` to revert and expect relaying entities to run the sufficient amount of checks and simulations to prevent this risk.

Documentation has been added to outline this expectation: PR 51.

Spearbit: Acknowledged.

5.4.7 WebAuthn verifiers implicitly assume deployment of P256 signature validators in current chain

Severity: Informational

Context: UserOpWebAuthnCosignVerifier.sol#L44, UserOpWebAuthnVerifier.sol#L36

Description: The `UserOpWebAuthnVerifier` and `UserOpWebAuthnCosignVerifier` contracts rely on the `WebAuthn` library from Solady, which uses the `P256` library for secp256r1 signature verification. The `P256` library expects specific contracts to be deployed at predetermined addresses to function correctly. This deployment dependency is not documented in the verifier contracts, potentially causing unexpected runtime errors if these precompiles are unavailable.

Recommendation: Document the deployment requirements directly in the WebAuthn verifier contracts. This documentation will help integrators understand the infrastructure requirements before attempting to use these verifiers.

Stackup: Fixed in PR 30.

Spearbit: Fix verified.

5.4.8 What if a roothash update fails on one of the chains?

Severity: Informational

Context: Keystore.sol#L17, Keystore.sol#L28-L31

Description: Assume a roothash update fails on one of the chains, for example if a `verifier::validateData()` fails, for example due to a bug. In that situation the update won't be processed on that chain and the nonce will not be increased. Also future updates won't be processed. In theory a new update could be created with the same nonce, however then there are two updates with the same nonce, which has to be coordinated for deployments on new chains.

Recommendation: In the offchain code: be prepared to handle this situation and be able to apply the approriate update on new chains.

Stackup: Fixed in PR 56.

Spearbit: Fix verified.

5.4.9 Native Solidity slicing can be used

Severity: Informational

Context: Keystore.sol#L49-L51

Description: In function `registerNode()` the variable `node` is of type `calldata`, which means the native solidity slicing can be used, which is shorter.

Recommendation: Consider changing the code to:

```
- require(address(bytes20(LibBytes.slice(node, 0, 20))) != address(0), InvalidVerifier());
+ require(address(bytes20(node[0:20])) != address(0), InvalidVerifier());
```

Stackup: Fixed in PR 31.

Spearbit: Fix verified.

5.4.10 Missing short-circuit in dual signature verification

Severity: Informational

Context: UserOpWebAuthnCosignVerifier.sol#L43-L44

Description: The `UserOpWebAuthnCosignVerifier::validateData` function performs both cosignature and WebAuthn signature validation unconditionally. When the cosignature validation fails (that is, `cosignValid` is `false`), the function still executes the WebAuthn verification even though the overall validation will inevitably fail. This continues unnecessary work when early termination could immediately return the failure result.

Recommendation: Consider implementing fail-early logic to exit validation when the first signature check fails.

Stackup: This is intentional in order to maintain consistent gas usage between simulation with dummy signers and execution with actual signers. Added comment in PR 32.

Spearbit: Acknowledged.

5.4.11 Function `getRegisteredNode()` could be simplified

Severity: Informational

Context: Keystore.sol#L60-L66

Description: Function `getRegisteredNode()` is used to check off-chain if a particular node is cached or not. Currently it passes the entire `node` as a parameter, but it would be more logical to pass the `nodeHash`.

Recommendation: Consider changing the code to:

```
-  function getRegisteredNode(bytes32 refHash, address account, bytes calldata node)
+  function getRegisteredNode(bytes32 refHash, address account, bytes32 nodeHash)
       // ...
-      return _nodeCache[_getCurrentRootHash(refHash, account)][keccak256(node)][account];
+      return _nodeCache[_getCurrentRootHash(refHash, account)][nodeHash][account];
```

Stackup: Fixed in PR 33.

Spearbit: Fix verified.

5.4.12 20 could be a constant

Severity: Informational

Context: Keystore.sol#L50-L51, Keystore.sol#L104-L107, Keystore.sol#L121

Description: The value 20 is frequently used but its meaning is not directly obvious.

Recommendation: Consider using a constant with a meaningful name.

Stackup: Fixed in PR 34.

Spearbit: Fix verified.

5.4.13 Unnecessary signature validations when signature count is below threshold

Severity: Informational

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

Description: The `UserOpMultiSigVerifier::validateData` function processes all provided signatures even when there are insufficient signatures to meet the threshold. When `signatures.length < threshold`, the validation will inevitably fail since there aren't enough signatures provided. However, the function still processes all signatures through the validation loop.

Recommendation: Consider adding a length check before signature validation to immediately fail when insufficient signatures are provided.

Stackup: We are optimising for the happy path of a `UserOperation`. If signature count is below threshold it will still fail and get rejected by the bundlers during simulation before making it onchain. During actual onchain validation we can more or less assume its going to pass and the gas spent on the conditional check is not necessary.

Spearbit: Acknowledged.

5.4.14 Unnecessary `else`

Severity: Informational

Context: Keystore.sol#L103-L105

Description: The `else` in function `_unpackNode()` isn't necessary because its right after a revert. Removing it makes the code more readable.

Recommendation: Consider removing the `else`.

Stackup: Fixed in PR 35.

Spearbit: Fix verified.

5.4.15 Function name of `_validateNode` doesnt cover its functionality

Severity: Informational

Context: Keystore.sol#L113

Description: The function `_validateNode()` also retrieves `node` information from the cache if necessary. This isn't clear from the function name.

Recommendation: Consider changing the name to also indicate it retrieves from cache.

Stackup: Fixed in PR 36.

Spearbit: Fix verified.

5.4.16 Use named parameters for nested mappings

Severity: Informational

Context: Keystore.sol#L13-L15

Description: Solidity version 0.8.18ˆ has the capability for named parameters which greatly enhances code readability of nested mappings.

Recommendation: Consider using named parameters such as `mapping(bytes32 refHash => mapping(address account => bytes32 rootHash))`.

Stackup: Fixed in PR 37.

Spearbit: Fix verified.

5.4.17 Replayability as default should be explicitly documented

Severity: Informational

Context: Keystore.sol#L42-L47

Description: `isValidSignature()` is a wrapper for the verifiers actual validation through `validate()` which should be free to check for nonces or not, depending on functionality, as replayability might be desired in some scenarios. However, nonce verification is usually the default expected as many ERC's such as 4337 and 2612 enforce it. In this case verifiers do not implement any nonce verification and allow for signature replayability by default.

Recommendation: Having replayability as default should be explicitly documented in all verifiers.

Stackup: Fixed in PR 53.

Spearbit: Fix verified.

5.4.18 NFTs can't be stored in `KeystoreAccount`

Severity: Informational

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

Description: NFTs can't be stored in `KeystoreAccount` because it doesn't have the callbacks `onERC721Received()`, `onERC1155Received()` and `onERC1155BatchReceived()`. Several DeFi primitives use NFTs.

Recommendation: Consider supporting the callbacks `onERC721Received()`, `onERC1155Received()` and `onERC1155BatchReceived()`.

Stackup: Fixed in PR 38.

Spearbit: Fix verified.

5.4.19 `sd.index` could be too large

Severity: Informational

Context: UserOpMultiSigVerifier.sol#L28, UserOpMultiSigVerifier.sol#L47-L51

Description: In the for loop in `validateData()`, `sd.index` could be too large which would lead to a revert, which might be difficult to troubleshoot.

Recommendation: Consider checking `sd.index < signatures.length`.

Stackup: We are optimising for the happy path of a `UserOperation`. Assuming bundlers have not rejected the transaction during simulation then actual onchain validation is likely to pass and the conditional check would be a waste of gas.

Spearbit: Acknowledged.

5.4.20 Difference between spec and code

Severity: Informational

Context: spec.md#L82-L87, spec.md#L92-L95, spec.md#L120, Keystore.sol#L13, Actions.sol#L4-L9, Actions.sol#L14-L17

Description: Contract `Actions` defines proofs as `bytes proof` however the specification uses `bytes32[] proof`. Contract `Keystore` defines `mapping ... internal _rootHash`, while the specification uses `mapping ... public rootHash`.

Recommendation: Consider updating the specification.

Stackup: Fixed in PR 39.

Spearbit: Fix verified.

5.4.21 `refHash` might not be unique

Severity: Informational

Context: KeystoreAccountFactory.sol#L23

Description: The `refHash` doesn't have to be unique (for instance if you want to create multiple account with the same signers). In that situation it should be detected that the creation fails and different salt should be used.

Recommendation: Consider documenting how to handle deployments with the same `refHash`.

Stackup: Fixed in PR 40.

Spearbit: Fix verified.

5.4.22 Different proof type used in `registerNode()`

Severity: Informational

Context: KeystoreAccount.sol#L54, Keystore.sol#L49, Actions.sol#L9, Actions.sol#L17

Description: Function `registerNode(..., bytes32[] calldata proof,...)` is the only function that used a `bytes32[]` format for a proof. All other places use `bytes` for the proof.

Recommendation: Consider changing `registerNode()` to use `bytes proof` and do the `abi.decode()` inside of the function.

Stackup: We are okay with keeping this as is. Using `bytes proof` means more gas spent on `abi.decode` and not being able to use `verifyCalldata` (which was implemented for "`verifyCalldata()` can be used").

The alternative of changing `action.proof` to `bytes32[]` also won't work due to the caching logic.

Spearbit: Acknowledged.

5.4.23 Underspecified `data` parameter in verifier contracts

Severity: Informational

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

Description: The `validateData` functions in verifier contracts lack documentation for the expected format of the `bytes data` parameter. Each verifier expects different data structures, but this isn't stated in the specification file nor in the functions' docstrings.

The current undocumented formats are:

- `UserOpECDSAVerifier`: Raw ECDSA signature OR ABI-encoded `PackedUserOperation`.
- `UserOpWebAuthnVerifier`: WebAuthn signature data OR ABI-encoded `PackedUserOperation`.

- `UserOpMultiSigVerifier`: `SIGNATURES_ONLY_TAG + SignerData[]` OR ABI-encoded `PackedUserOperation`.
- `UserOpWebAuthnCosignVerifier`: Similar to WebAuthn but with cosigner data.

Recommendation:

1. Add function documentation, including comprehensive docstrings for each verifier's `validateData` function specifying expected data formats.
2. Update `spec.md` with a dedicated section documenting each verifier's parameter format expectations.
3. Document dual-mode pattern, explaining the common `PackedUserOperation` vs. "raw" data handling pattern used across verifiers.

Stackup: Note that `spec.md` was intentionally not edited to include details for each implemented verifier. The `spec.md` doc is intended to explain the Keystore and generalised interfaces. Since many types of verifiers can be implemented we don't think this is the right place for such documentation. Fixed in PR 41.

Spearbit: Fix verified.

5.4.24 Bad actor could hide a malicious configuration in the Merkle tree

Severity: Informational

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

Description: The `KeystoreAccount` stores only Merkle tree root hashes on-chain while keeping the actual configuration trees off-chain. This saves gas and aims to enhance configuration privacy, but it means users can't see all the ways their account can be accessed.

When deploying a `KeystoreAccount`, users create a Merkle tree with one or many verification methods (like multisig, single signatures, WebAuthn, etc.) but only store the root hash of the configuration on-chain. When accessing the account, they provide a proof showing that their specific verification method exists in the tree. The system validates this proof against the stored root hash, but it never reveals what other verification methods might exist in the same tree.

This creates an interesting trust dynamic. The person who crafts the first call to `handleUpdates` gets to set the reference hash that defines the initial configurations that are valid for that account. Everyone else has to trust that this entity accurately represented the complete intended set of verification methods. If someone claims an account is a 3-of-5 multisig, there's no way to verify that they didn't also include a single signature that they control.

The system actually encourages multiple verification methods in the same account. Users might legitimately want different authentication for daily operations versus emergency recovery, or they might want to gradually upgrade from simple ECDSA to more sophisticated schemes. The privacy model means recovery guardians stay hidden until actually needed, which sounds a sensible design decision.

But there's a trade-off. Users can verify that specific configurations exist in the tree, but they can't prove that other configurations don't exist. The Merkle proof shows inclusion, not completeness.

This is all intended behavior, although it might not be obvious for regular users. The development team has chosen to favor privacy and gas efficiency, over account transparency. That's a reasonable choice for multiple use cases, but it's important users understand the whole picture. In high-trust situations where multiple parties are involved, it's advised they find ways to verify the full configuration set outside the protocol itself.

The practical impact depends on the situation. Account creators need to be transparent with other users about what verification methods they've included. Users of someone else's account need to trust them or find ways to verify the configuration independently. For business accounts with multiple stakeholders, processes are needed to audit and verify configurations that go beyond what the protocol provides.

The security model assumes account creators act honestly and communicate their configurations appropriately. The on-chain validation methods prevents anyone from using verification methods that weren't committed to the tree, but it doesn't prevent someone from committing to more methods than they advertised.

This is a sensible design for use cases where privacy matters and users can establish appropriate trust relationships. The privacy comes with trade-offs, and planning is required for situations where stronger guarantees about configuration completeness are needed.

Recommendation: Have a way to prove how the merkletree is built, that way you can see nothing is added.

Stackup: Fixed in PR 54.

Spearbit: Fix verified.

5.4.25 Smart account implementations must implement signature replay protection

Severity: Informational

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

Description: `Keystore.validate` does not include any signature replay protection:

```
function validate(ValidateAction calldata action) external view returns (uint256 validationData) {
    (, bytes memory node) = _validateNode(action.refHash, msg.sender, action.proof, action.node);

    (address verifier, bytes memory config) = _unpackNode(node);
    return IVerifier(verifier).validateData(action.message, action.data, config);
}
```

As such, it's important that any smart account implementation using the `Keystore` implements relevant signature replay protection to safely interact with it.

Recommendation: Include clear documentation indicating that signature replay protections should be implemented either by.

- Smart account implementations by including checks on the chainId, account, verifying contract, nonce, etc. in the signed `action.message`.
- Verifier implementations by also checking contents of `action.message` .
- Other external contracts which implement such checks after calling `isValidSignature()`.

Stackup: Fixed in PR 53.

Spearbit: Fix verified.

5.4.26 Adding new verifiers has to be controlled

Severity: Informational

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

Description: Adding new verifiers has to be controlled very well. A verifier that always returns `SIG_VALIDATION_SUCCESS` will allow a full takeover of the account.

Recommendation: Verifiers should not be allowed to be added permissionless. They should be very well reviewed/audited before being added.

Stackup: Fixed in PR 55.

Spearbit: Fix verified.