Quantstamp DRAFT

Openfort - 7702 Smart Account

Executive Summary

This audit report was prepared by Quantstamp, the leader in blockchain security.

Type	ERC-4337 and EIP-7702 Smart Account
Timeline	2025-07-31 through 2025-08-12
Language	Solidity
Methods	Architecture Review, Unit Testing, Functional Testing, Computer-Aided Verification, Manual Review
Specification	None
Source Code	• openfort-xyz/openfort-7702-account #3682208
Auditors	• Ruben Koch Senior Auditing Engineer • Rabib Islam Senior Auditing Engineer • Tim Sigl Auditing Engineer • Nikita Belenkov Senior Auditing Engineer

Documentation quality	Medium
Test quality	Medium
Total Findings	16 Unresolved: 16
High severity findings	1 Unresolved: 1
Medium severity findings	1 Unresolved: 1
Low severity findings	7 Unresolved: 7
Undetermined severity findings	0
Informational findings	7 Unresolved: 7

Summary of Findings

In this audit we assessed Openfort's EIP-7702 smart account implementation with full ERC-4337 compatibility, upgradability and session keys support.

It supports multiple key types (EOA, WebAuthn, P-256/P-256NONKEY) with granular controls such as: tx limits, ETH/token quotas, function selector filtering, and contract whitelisting. Social recovery is built in via guardian proposals, timelocks and quorum-based approval. The account supports two ERC-7821 execution modes: Single batch and batch of batches.

Overall, the design is feature-rich and well implemented but would benefit from additional restrictions on session keys and hardening around their key configuration. As pointed out in OPF-1 there are no gas restrictions on a session key and it can withdraw the account's funds at the `EntryPoint`. In addition, there are several possibilities to misconfigure session keys (e.g. OPF-4, OPF-5) which can have significant impact, such as being able to transfer ERC-20 tokens without permission. We also want to highlight a potential breach of permission separation in the ERC-1271 `isValidSignature()` in combination with ERC-2612 permit flows (see OPF-2).

ID	DESCRIPTION	SEVERITY	STATUS
OPF-1	Session Key Owners Can Drain Native Assets	High	Unresolved
OPF-2	Approvals to Contracts Designed to Facilitate Transfers and Approvals Using `isValidSignature()` Can Cause Unintended Access of Funds	Medium	Unresolved
OPF-3	Incomplete Key Removal And Dangers of Reactivated Keys	Low	Unresolved
OPF-4	Session Keys Cannot Access Extended Token Functions	Low	Unresolved
OPF-5	Session Keys From Previous Delegations are not Cleared and Remain Valid	Low	Unresolved

ID	DESCRIPTION	SEVERITY	STATUS
OPF-6	Dangerous Combination of Whitelist and Token Spend Limit	Low	Unresolved
OPF-7	Gas Griefing by the Bundler Is Possible Due to Unbound Signature Size Decoding	Low	Unresolved
OPF-8	Mixing of Name-Spaced Storage Layout and Custom Storage Layout Could Cause Unintended Behaviour	Low	Unresolved
OPF-9	Improved Input Validation For Key Registrations	Low	Unresolved
OPF-10	Account Incorrectly Returns ERC-7821's `mode=2` as Supported	Informational	Unresolved
OPF-11	Incorrect Behaviour Around `SIG_VALIDATION_FAILED`	Informational	Unresolved
OPF-12	Improvements to ERC-1271 Support	Informational	Unresolved
OPF-13	Incorrect EIP-712 Implementation	Informational	Unresolved
OPF-14	All Parameters of `initialize()` Should Be Signed	Informational	Unresolved
OPF-15	Failing Execution Phases Will Still Cause Limits of Session Keys to be Consumed	Informational	Unresolved
OPF-16	Signature Type Interchangeability Can Reduce Security	Informational	Unresolved

Assessment Breakdown

Quantstamp's objective was to evaluate the repository for security-related issues, code quality, and adherence to specification and best practices.

> **Disclaimer**
> Only features that are contained within the repositories at the commit hashes specified on the front page of the report are within the scope of the audit and fix review. All features added in future revisions of the code are excluded from consideration in this report.

Possible issues we looked for included (but are not limited to):

- Transaction-ordering dependence
- Timestamp dependence
- Mishandled exceptions and call stack limits
- Unsafe external calls
- Integer overflow / underflow
- Number rounding errors
- Reentrancy and cross-function vulnerabilities
- Denial of service / logical oversights
- Access control
- Centralization of power
- Business logic contradicting the specification
- Code clones, functionality duplication
- Gas usage
- Arbitrary token minting

Methodology

1. Code review that includes the following
   1. Review of the specifications, sources, and instructions provided to Quantstamp to make sure we understand the size, scope, and functionality of the smart contract.
   2. Manual review of code, which is the process of reading source code line-by-line in an attempt to identify potential vulnerabilities.
   3. Comparison to specification, which is the process of checking whether the code does what the specifications, sources, and instructions provided to Quantstamp describe.
2. Testing and automated analysis that includes the following:

    1. Test coverage analysis, which is the process of determining whether the test cases are actually covering the code and how much code is exercised when we run those test cases.
    2. Symbolic execution, which is analyzing a program to determine what inputs cause each part of a program to execute.
3. Best practices review, which is a review of the smart contracts to improve efficiency, effectiveness, clarity, maintainability, security, and control based on the established industry and academic practices, recommendations, and research.
4. Specific, itemized, and actionable recommendations to help you take steps to secure your smart contracts.

Scope

Files Included

- src/core/BaseOPF7702.sol
- src/core/Execution.sol
- src/core/KeysManager.sol
- src/core/OPF7702.sol
- src/core/OPF7702Recoverable.sol
- src/core/OPFMain.sol
- src/interfaces/IKey.sol
- src/interfaces/iOPF7702Recoverable.sol
- src/libs/KeyDataValidationLib.sol
- src/libs/KeyHashLib.sol
- src/libs/UpgradeAddress.sol
- src/libs/ValidationLib.sol
- src/utils/ERC7201.sol
- src/utils/SpendLimit.sol
- src/utils/WebAuthnVerifierV2.sol

Operational Considerations

- Upgradeability: The smart account is expected to be deployed via two pathways: (1) via delegation directly to the implementation; and (2) via delegation to a proxy that delegates to the implementation. In pathway (2), the proxy admin may upgrade the implementation contract without the smart account users' knowledge. Moreover, such upgrades may include new vulnerabilities that compromise the security of the users' funds.
- Delegation Target: Only delegate (EIP-7702) to audited implementations; malicious implementations can poison persistent mappings (keys, usedChallenges). Verify bytecode before (re)delegation.
- If used with ephemeral EOA, ensure only delegating to contracts implementing `upgradeProxyDelegation()` to prevent delegation locking.
- Session Key Scope: Ensure limits (tx count, ETH/token caps, selector/target whitelists) are minimally sufficient; avoid granting execution patterns that can self-upgrade or reconfigure guardians.
- Always enforce a contract whitelist on session keys.
- Guardian Quorum: Assign a sufficient number of guardians to minimize risk of collusion and avoid a single point of failure.
- Verify new WebAuthnVerifier and EntryPoint implementations before switching to them.
- Token spend limits may not work with non-ERC-20 tokens.

Key Actors And Their Capabilities

Delegating EOA (EIP-7702 self-call)

Responsibilities

- Initialize account; set EntryPoint/WebAuthnVerifier; manage guardians; cancel account recovery; redelegate code; execute unrestricted transactions; register/revoke session keys.

Trust Assumptions

- Fully trusted; capabilities similar to master key. Allowed by `_requireForExecute()` since `msg.sender == address(this)` under 7702.

Master Key

Responsibilities

- Everything the delegating EOA can do, except initialization of the account.

Trust Assumptions

- Fully trusted; compromise results in full loss of control.

Session Keys (EOA/WebAuthn/P256/P256NONKEY)

Responsibilities

- Authorize execution within policy: validity window, tx count, ETH/token limits, whitelists/selectors; no self-calls allowed.

Trust Assumptions

- Partially trusted or untrusted; impact bounded by key constraints and configuration.

Guardians

Responsibilities

- Start account recovery.

Trust Assumptions

- Majority quorum required; sub-threshold collusion tolerated; threshold compromise can reassign account control.

Findings

OPF-1 Session Key Owners Can Drain Native Assets

• High ⓘ Unresolved

Description: Session keys essentially enable a semi-trusted or untrusted entity to operate with limited access directly on the account. Numerous checks are added to ensure that a session key does not perform too many calls, does not transfer too much of the native asset and does not exceed a spending limit on a designated ERC20 token. While ERC-4337 provides a framework for including Paymasters that cover a user's gas costs, the standard case is that a user essentially pays the Bundler for their service, including the UserOperation's gas costs, with their EntryPoint deposits, which users authorize by providing some form of a signature that passes the account's validation. However, since session key owners also have a way to pass the validation on the account they are registered in, they can unlock those EntryPoint deposits. As there are neither gas limit checks nor a gas budget for a session key, the session key can sign a UserOperation that drains the account's entire deposit. Since the role of the Bundler is permissionless, a malicious session key owner could bundle such a UserOperation themselves, directly profiting from the attack.

Exploit Scenario:

1. Alice authorizes a session key for Bob with any `ethLimit` and `spendTokenInfo.limit`. For the sake of an example, suppose Bob has been authorized to spend 100 USDC.
2. Bob crafts a UserOperation with a payload that doesn't spend any ETH or tokens, but with extremely high gas limits or gas price – for example, setting `callGasLimit` and `preVerificationGasLimit` to 1000000 with `maxFeePerGas` set to a very high value.
3. Bob submits the UserOperation to the EntryPoint contract.
4. EntryPoint calls `validateUserOp()` on Alice's account. Since the session key's signature is valid and all the payload-related checks are passing, the validation succeeds.
5. EntryPoint calculates the required prefund based on the UserOperation's high gas limits and deducts this large amount from the victim's deposit.
6. EntryPoint executes the UserOperation. The actual call may use very little gas, or even revert.
7. EntryPoint calculates the final gas cost and pays Bob, who was the bundler for this UserOperation.
8. Bob can profit from either a too high gas price since the bundler keeps the spread between the declared gas price in the UserOperation and the actually paid gas price or from a too high gas limit because there is a 10% fee on the unused gas which is also collected by the bundler.

Recommendation: Session keys should always have an associated gas budget. Alongside updates to the quota and other limits, this gas budget should be reduced based on the UserOperation's defined gas costs. See this implementation as a reference for gas accounting for session keys. Also note the special handling in case of Paymaster usage.

OPF-2 Approvals to Contracts Designed to Facilitate Transfers and Approvals Using `isValidSignature()` Can Cause Unintended Access of Funds

• Medium ⓘ Unresolved

File(s) affected: `OPF7702.sol`

Description: ERC-1271 allows smart contracts to validate signatures using logic specific to the contract. In the case of `OPF7702`, the function validates the signatures of most keys associated with the contract, including session keys that are registered, live (meaning currently valid), and have not exceeded their `Call` quota. This is regardless of the signature they sign, and does not at all involve their individual policies which include whitelisted contracts, spend limits for ETH and tokens, and permitted selectors.

In user space, it is relatively common to approve contracts that use `isValidSignature()` to perform gasless transactions on behalf of the user. For instance, Uniswap users may approve the Permit2 contract. Afterwards, users may sign signatures that allow the Uniswap application to execute transactions on the users' behalf via signatures sent offchain in order to swap tokens. When these signatures are sent from accounts without code, the normal ECDSA flow is used to verify signature validity. However, when the user is operating with a 7702 smart account, their account will carry code, and the Permit2 contract will instead verify the signature using the smart account's implementation of `isValidSignature()`.

However, contracts such as Permit2 may be used more generally: for example, the Permit2 contract allows the user to sign not just for swaps, but also for arbitrary approvals and transfers. These actions will then be executed onchain on the behalf of the signer.

When the `masterKey` owner registers a key on the Openfort wallet, they are likely to carry the reasonable assumption that the capabilities of the new signer will be constrained to interacting with whitelisted addresses and only using the selectors approved by the owner. However, consider the following scenario:

1. Alice, the `masterKey` holder, registers a session key for a third party, Bob, that can only transfer up to 100 USDC from the wallet.

2. Alice approves the Permit2 contract to spend 1 WETH (or, even more common, gives max approval), hoping to swap it for MKR.
3. Bob, noticing the approval to Permit2, calls `Permit2.permitTransferFrom()` with a signature that passes `isValidSignature()` validation, sending the 1 WETH to his own personal wallet.

Recommendation: Only allow the masterKey and wallet owner to be able to sign valid signatures for use via `isValidSignature()`. If it is desired for the contract to also validate session key signatures, implement a new function with a different selector for this purpose.

OPF-3
Incomplete Key Removal And Dangers of Reactivated Keys

• Low ⓘ Unresolved

File(s) affected: `KeysManager.sol`

Description: Upon deactivation of a key via `KeysManager._revokeKey()`, several fields are not properly reset or removed:

- The `pubKey` field associated with a key is not removed upon deactivation.
- The `whitelisting` boolean and the associated `whitelist` mapping are not removed either. While the mapping itself can not be deleted in Solidity, the presence of the flag and residual mapping entries can lead to unintended authorizations.

Upon reactivation of a previous key with a whitelist, the old whitelist will re-activate, even if the `_keyData` does not explicitly include those addresses in the whitelist. As keys may be registered to be controlled by untrusted parties, this behavior poses a security risk.

Recommendation: Delete the `pubKey` field in the `_revokeKey()` function. Prevent reactivation of any key that had whitelisting enabled previously. So, leave the `whitelisting` flag for deactivated keys and revert if a key is attempted to be re-registered that already has this value set in storage. Alternatively, explicitly document this behaviour, so end users can make an informed decision to reactivate a key when the exact old whitelist is known and explicitly intended to be reused.

To enable a proper cleaning of state, an `EnumerableSet` could be used for the whitelist, which offers iterable and deletable sets, enabling full removal on key revocation.

OPF-4 Session Keys Cannot Access Extended Token Functions

• Low ⓘ Unresolved

File(s) affected: `OPF7702.sol`

Description: The function `_validateTokenSpend()` reads the last 32 bytes of a session key's calldata and attempts to subtract it from the key's `spendTokenInfo.limit`. However, this implicitly places unpredictable constraints on the kinds of functions that a session key can reasonably execute. For example, calling a function like `redeem(uint256, address, address)` on an ERC4626 tokenized vault may result in unexpected outcomes, like the spend limit being deducted by an unforeseen amount. However, this behavior is not documented.

Recommendation: If it is intended that session keys only call functions like `approve()` and `transfer()`, the signatures of which end in `uint256`, then this should be documented, and the selectors available to session keys should be limited.

OPF-5
Session Keys From Previous Delegations are not Cleared and Remain Valid

• Low ⓘ Unresolved

File(s) affected: `BaseOPF7702.sol`

Description: The `_clearStorage()` function resets specific fixed storage slots but cannot remove entries from mappings. Consequently, critical mappings such as `idKeys`, `keys`, and `usedChallenges` persist across delegations. A malicious delegated contract can deliberately insert privileged entries into these mappings. When control is redelegated to the legitimate Openfort account contract, the injected entries are implicitly trusted and may be used to validate UserOperations without detection.

While a custom storage layout prevents accidental key collisions, targeted and intentional mapping insertions remain feasible. For high-value accounts, an attacker has a strong incentive to phish EOAs into delegating to a malicious contract capable of inserting keys at known mapping locations.

Exploit Scenario:
1. Victim delegates their EOA to a malicious contract.
2. The malicious contract inserts a privileged key (e.g., `sKey.masterKey == true`) into the `keys` mapping at the expected storage location.
3. The victim later redelegates back to the `OPFMain` contract.
4. During UserOperation validation, the malicious key is retrieved from storage and passes the validation checks.

Recommendation: Implement one of the following recommendations, noting that these are conceptual approaches which should be tested and validated to ensure they work as intended in the current architecture:

- **Randomized Storage Base Slot** – Incorporate a **non-deterministic** storage slot salt into the calculation of the base storage slot during account initialization (e.g., `keccak256("openfort.baseAccount.7702.v1" + salt)`). This ensures the mapping storage layout is unpredictable and prevents pre-insertion attacks.

- **Key Registration Gating** – The `KeysManager.id` is cleared on initialization. Therefore you could enforce an invariant that retrieved keys during validation are `retrievedKey.id <= KeysManager.id`. The storage slot has to be depended on the `id` so only one key per `id` can be stored. This check ensures keys are registered during the current delegation and are not pre-inserted into the storage.

OPF-6
Dangerous Combination of Whitelist and Token Spend Limit

• Low ⓘ Unresolved

File(s) affected: `OPF7702.sol`

Description: If a token spend limit is defined for a session key, it requires the necessary selectors to be set (e.g. `approve()` or `transfer()`) for validation to pass. As the selector list is shared throughout the whitelist, if a second ERC20 token that is not the `SpendTokenInfo.token` were to "just" be added to the key's whitelist, the key could consume all the ERC-20 tokens. This, of course, would require significant misconfiguration, but still remains a possibility

Recommendation: Consider moving an individual approved selector array to the `SpendTokenInfo` struct and check for it being used as part of the `_validateTokenSpend()` validation function. This would make the common "spending selectors" only accessible to the `SpendTokenInfo.token` address instead of to the entire whitelist.

OPF-7
Gas Griefing by the Bundler Is Possible Due to Unbound Signature Size Decoding

• Low ⓘ Unresolved

Description: The `_validateSignature()` function uses unconstrained `abi.decode()` to parse the signature parameter, which allows the bundler to append arbitrary data after the expected `sigData` parameter. Since abi.decode() drops trailing data, these extra bytes go undetected but still consume gas (for example, `abi.encode(key, bytes, stuffed=bytes)` can decode cleanly to `(KeyType, bytes)` while silently dropping the stuffed bytes. So it is also important to validate the length of the `userOp.signature` and not only the length of the decoded signature.

The bundler controls the signature input and gets compensated based on gas usage, they can maximize their profit by including extra data that needs to be processed during decoding, leading to higher gas costs for the user.

Exploit Scenario:

1. A user submits a valid UserOperation with a legitimate WebAuthn signature
2. The bundler appends additional bytes to the signature data
3. When the WebAuth signature is decoded, it processes this input using `abi.decode()`, it consumes extra gas to process the padded data
4. The user pays more in gas fees than necessary, and the bundler profits from this overhead

While this is bounded by maxFeePerGas, it still allows the bundler to maximize their profit at the user's expense

Recommendation: Implement a length check on the `userOp.signature`. For variable-length signatures, such as those used in WebAuthn, the expected signature length must be computed based on the properties of the signature.

```
(KeyType sigType, bytes memory sigData) = abi.decode(userOp.signature, (KeyType, bytes));
expectedSignatureLength = dependingOnKeyTypeCalculateSignatureLength(sigType);
require(userOp.signature.length == expectedSignatureLength, "Invalid signature length");
```

OPF-8
Mixing of Name-Spaced Storage Layout and Custom Storage Layout Could Cause Unintended Behaviour

• Low ⓘ Unresolved

Description: The account uses a custom storage location to avoid storage collisions from prior 7702 delegations of an EOA. However, ERC-7201 namespaced storage is also included through the `EIP712Upgradeable` and `ReentrancyGuardUpgradeable` libraries. Namespaced storage would be shared between different delegations, as they specify an exact storage slot, not a custom offset. Therefore, prior delegations might have overridden the account storage on the EOA, mixing storage from prior usage with this account. That could cause incorrect domain separators to be build, making the signature validation work non-deterministically throughout many accounts. More severely, it could stop EOAs from being able to call `initialize()` on their account, as in case the EOA has delegated to an initializable contract previously, the `initializer` modifier would revert due to the `InitializableStorage._initialized` variable having been set by prior delegations, stopping the possibility for them to add a Master key.

Recommendation: Use the non-upgradeable variants of the two libraries so that the two libraries adapt to the custom storage layout. Care should be taken, as this might adjust the overall storage layout.

OPF-9 Improved Input Validation For Key Registrations

• Low ⓘ Unresolved

File(s) affected: `OPF7702Recoverable.sol`, `KeysManager.sol`

Description: Further input validation should be performed to reduce the possibility of end users accidentally creating invalid session key state:

- All keys should either populate the `eoaAddress` field or the `pubKey` struct field, not possibly both. We would recommend removing the `DEAD_ADDRESS` placeholder value in case the `pubKey` field is intended to be populated. Alternatively, check for it explicitly in that case.
- All non-master session keys should have enforced that `whitelisting = true`.
- Master keys should only set a `validUntil = type(uint48).max`.
- Validate `KeyData.masterKey` is always `true` for a master key and the key is active.
- All other fields, except the `pubKey` should remain zero.
- Master keys added in `OPF7702Recoverable.sol.completeRecovery()` and `OPF7702Recoverable.sol.initialize()` should be validated according to the rules defined above in a separate function.
- Make sure zero address cant be added to whitelists, as else a a attack vector of a self-call bypass gets a protection layer removed

Recommendation: Consider adding these input validations.

OPF-10 Account Incorrectly Returns ERC-7821's `mode=2` as Supported

• Informational ⓘ Unresolved

File(s) affected: `OPF7702.sol`, `Execution.sol`

Description: The `Execution.supportsExecutionMode()` function returns all modes 1,2 and 3 defined by ERC-7821 as supported. However, the `OPF7702._validateExecuteCall()` function, invoked before any execution, only passes for `mode = 1` or `mode = 3`

Recommendation: Don't return `mode = 2` as supported in the `supportsExecutionMode()` function, or alternatively implement it properly.

OPF-11 Incorrect Behaviour Around `SIG_VALIDATION_FAILED`

• Informational ⓘ Unresolved

Description: According to ERC-4337, `SIG_VALIDATION_FAILED` should only be returned in case of signature mismatches, all other cases should revert. Also, the idea around the `SIG_VALIDATION_FAILED` mechanic is to enable gas estimation with dummy signatures for bundlers. Therefore, cases leading to `SIG_VALIDATION_FAILED` should not return early, but instead update a variable to that value that is ultimately returned at the very end of the function.

Recommendation: * Remove the early returns for `SIG_VALIDATION_FAILED` as described

- In `OPF7702._validateSignature()` a `revert()` should be implemented in case none of the three `sigTypes` were used.
- In the `validateKeyTypeX()` functions, a check for `signer == address(0)` is unnecessary with the used version of the OpenZeppelin contracts. That case will, as it is correct, automatically revert. Hence, those `if` -blocks can be removed. The rest of the `SIG_VALIDATION_FAILED` cases in that function are correct.
- In `_validateKeyTypeWEBAUTHN()`, a `revert()` should be implemented in case `usedChallenges[userOpHash] == true` instead of the current return of `SIG_VALIDATION_FAILED`.
- The library for the WebAuthn and P256 signature validation does not expose a proper differentiation between malformed and mismatched signatures. We recommend to stick to `SIG_VALIDATION_FAILED` for all cases as it is.
- *

As `ecrecover()` takes care of all malformed sigs, so only leaves signature mismatch, I think the only wrong case of this is L106, which however should never hit in the first place, as that's gonna be a revert within the OZ library

OPF-12 Improvements to ERC-1271 Support

• Informational ⓘ Unresolved

Description: For ERC-1271 `isValidSignature()` checks, `0xffffffff` should be returned for any form of incorrect signatures. However, in `OPF7702._validateEOASignature()`, `ECDSA.recover()` is used, which reverts for any malformed signatures and also if the precompile derived the zero address as a signer. Therefore, the current check for `if (signer == address(0))` will never hit, as such a call would cause a revert.

Recommendation: Use the tryRecover() function instead and, in case of errors or incorrect signer, return `0xffffffff`.

OPF-13 Incorrect EIP-712 Implementation

• Informational ⓘ Unresolved

File(s) affected: `OPF7702Recoverable.sol`

Description: Minor improvements to the EIP-712 implementation have been identified:

- The `RECOVER_TYPEHASH` is encoded with a very different set of values as part of the initialization flow. There should be a separate `INIT_TYPEHASH = keccak256("InitializionData(uint256 x,uint256  y,address  eoaAddress,KeyType keyType,address _initialGuardian)");` for that aspect. Consider implementing the `InitializationData` struct accordingly and provide it as an input to the `initialize()` function.

- Parameter names in type hash declaration differ compared to the actual `RecoveryData` struct in its appropriate function at getDigestToSign()`.

Recommendation: Consider implementing proper separation between the type hashes as suggested.

OPF-14 All Parameters of `initialize()` Should Be Signed

• Informational ⓘ Unresolved

Description: As a best practice, all parameters of the `initialize()` function should be signed by the EOA, rather than a partial signing of all fields, especially since the key permissions of the master key are not signed. As there is strong access control in place to this function via the `requireForExecute()` check, this is mainly to adhere to best practices.

Recommendation: Consider including `_sessionKey` `_keyData` and `_sessionKeyData` in the digest.

OPF-15 Failing Execution Phases Will Still Cause Limits of Session Keys to be Consumed

• Informational ⓘ Unresolved

Description: In ERC-4337, state changes of the validation phase are not reverted on a reverting UserOperation's execution phase. As the session key's limit/quota consumptions are performed as part of the validation phases, this means that the key might have e.g. an ERC-20 spend limit reduction recorded, yet the actual execution of the transfer did not happen due to the revert.

Recommendation: We mainly want to raise awareness for this behaviour. Significant code changes would be required to adjust for this behaviour. A fix could be to route all UserOperations through `executeUserOp()`, the optional selector of accounts in ERC-4337, which forwards the full UserOperation struct in to the execution phase. While the validation for limits should still happen in the validation phase, this would enable setting the actual values in the execution phase, causing them to get rolled back too in case of a reverting execution phase. The ERC-7821 batching certainly introduces more complexity then, though, as potentially multiple calls adjust the same key's limits, which would be complex to monitor for in the validation phase without state updates.

OPF-16 Signature Type Interchangeability Can Reduce Security

• Informational ⓘ Unresolved

Description: The `keyId` computation in `KeyHashLib.sol` generates identical identifiers for different key types when they share the same public key coordinates (x, y). This means:

1. **Same Public Key, Different Security Models**: A key registered as `KeyType.WEBAUTHN` can later be used with `KeyType.P256` or `KeyType.P256NONKEY` signatures
2. **Security Guarantee Bypass**: WebAuthn provides additional security through:
   - Authenticator data verification (device attestation)
   - Client data validation (origin verification)
   - User verification flags (biometric/PIN requirements)
   - Challenge-response replay protection
3. **Signature Validation Routing**: The `_validateSignature()` function routes to different validation methods based on the signature's declared `KeyType`, not the registered key's type

Exploit Scenario:
An attacker could:

1. Register a key as `KeyType.WEBAUTHN` (with stricter attestation requirements)
2. Later use the same public key with a `KeyType.P256` signature
3. Bypass WebAuthn's security checks while maintaining the same permissions and limits

Recommendation: Implement stricter key type enforcement to prevent signature type interchangeability. More precisely, in the `_validateXSignature()` functions in the `OPF7702` contract, `X` being the intended signature validation flow, assure that in the `keyIds` mapping, the given public key's `keyHash` resolves to a `Key` struct with a `keyType` of the used signature validation.

Auditor Suggestions

S1 Improvements to the Setter Functions

Unresolved

File(s) affected: `BaseOPF7702.sol`

Description: In the `BaseOPF7702` contract, the `setEntryPoint()` and `setWebAuthnVerifier()` functions themselves emit an update event, as does the sub-call into the `UpgradeAddress` library. Furthermore, the event is not quite accurate if this is the first custom override of the field. In that case `oldEp` or `oldV` will remain zero in the event emission, yet the immutable values `ENTRY_POINT` or `WEBAUTHN_VERIFIER`, respectively, should be used. Furthermore, the provided, to-be-updated value can be the same one as previous value.

Recommendation: Remove one of the events in each of the affected cases. Consider adding input validation to not enable an identical override of the values.

S2 EntryPoint and WebAuthn Verifier Override Not Cleared

Unresolved

File(s) affected: `OPF7702Recoverable.sol`

Description: On calling `initialize()` the account's storage is mostly cleared to assure a fresh start. However, the namespaced storage for the EntryPoint and WebAuthn verifier are not overridden, meaning that possibly a prior custom configuration remains re-used, even after an attempted clearing of storage.

Recommendation: Clear the storage of those two variables in the `initialize()` function too.

S3 Notes on Recovery Period and Lock Period

Unresolved

File(s) affected: `OPF7702Recoverable.sol`

Description: The recovery mechanic uses a `recoveryPeriod` and a `lockPeriod`. Both periods start once a recovery process is initiated. The `recoveryPeriod` dictates when the recovery process can be completed. An ongoing recovery process can be checked for with the `requireRecovery(true)` function. In the `lockPeriod`, all guardian maintenance functions are paused.

All guardian state maintenance functions check consistently for the account to not be within a lock period. However, to a lesser extent, they also check for the account to not be within a recovery period. This latter property is only checked in the `confirmGuardian()` and `cancelGuardianProposal()` functions. It is not clear why only those two functions should check for there to not be an ongoing recovery. Our opinion is that once a recovery process is initiated, until the recovery process is completed or cancelled, there should not be any updates to the guardians' states.

The client clarified that `lockPeriod` is expected to be set to a greater value than `recoveryPeriod`, which would mean there is less room for guardian state changes in an ongoing recovery. However, as the check for an ongoing recovery period is missing in a number of the maintenance functions, if the recovery is not completed after the lock period, the recovery process could become invalidated due to state changes after the lock period, but before the recovery is finalized.

Recommendation: Consider unifying `recoveryPeriod` and `lockPeriod` and to only allow guardian maintenance in case of `requireRecover(false)`. If the two values are kept, enforce that `lockPeriod >> recoveryPeriod`.

S4 Incorrect Etching in the Test Suite

Unresolved

File(s) affected: `Keys.t.sol`, `Registration.t.sol`

Description: In all tests, the `setUp()` function initially etches (no 7702-vm.etch) into the implementation contract directly (without the 7702-prefix). This would essentially mimic regular, non-delegated deployment. This gets overridden in almost every test into a proper 7702-delegation, except in these two cases:

- `Keys.t.sol.tst_revokeALL()` does not re-etch to proper delegation-setup, meaning the tests operate as if it is a regular implementation contract.
- All tests in `Registartion.t.sol` do not re-etch to proper delegation-setup

Recommendation: Consider replacing the non-7702-prefixed `vm.etch` from the setup functions and do the proper delegation right away, instead of overriding it in every test.

S5 Improve Comments

Unresolved

File(s) affected: `BaseOPF7702.sol`, `OPF7702.sol`, `KeyValidationLib.sol`

Description: There are a number of instances where comments are either outdated or inaccurate.

1. At `BaseOPF7702.sol#L121` "Clear slots 8-14" should be "Clear slots 4-10"
2. The NatSpec for `isValidKey` makes mention of `_validateExecuteBatchCall()`, whereas that is not a defined or implemented function in the code.
3. `OPF7702.sol#L295,307` – it should be encoded as `execute(bytes32,bytes)`
4. `OPF7702.sol#L309` – it ensures that whitelisting is enabled AND that the target is whitelisted.
5. `KeyValidationLib.sol#L77` – the function returns `true` when the key is empty.

Additional comments should be added in order to enhance code clarity.

1. In IKey, the enum KeyType has EOA and WEBAUTHN as documented, but does not have P256 and P256NONKEY documented.
2. In SpendLimit, the assembly block in _validateTokenSpend() is insufficiently documented, as it does not explain the structure of innerData.

Recommendation: Update or add the listed comments appropriately.

S6 Gas Optimizations

Unresolved

File(s) affected: `OPF7702.sol`, `BaseOPF7702.sol`, `KeysManager.sol`

Description: 1. At `BaseOPF7702.sol#L112-114`, instead of calculating the slot during runtime, use a constant.
2. At `OPF7702.sol#L259`, use DeMorgan's Law to replace the clause with `(!(sKey.isRegistered() && sKey.isActive))`. The same applies at the following instances:
   1. `OPF7702.sol#L370`
   2. `OPF7702.sol#L492`
   3. `OPF7702.sol#L547`
   4. `OPF7702.sol#L585`
3. Use `unchecked` to subtract at `OPF7702.sol#L398`.
4. In `KeysManager._addKey()`, instead of ensuring that the `token` of `spendTokenInfo` is non-zero in an `if` and in the corresponding `else`, simply perform this check outside the `if-else` conditional.

S7 Suggestions for Code Improvements

Unresolved

Description: Some suggestions for code improvements have been identified:

- The `SpendLimit._validateTokenSpend()` function is overridden in the inheriting `OPF7702` contract with an almost fully identical, functionally equivalent implementation. Consider removing the implementation details of the function defined in the `SpendLimit` contract.
- Since Solidity version 0.8.22, unchecked increments of indexes from for-loops no longer offer any gas improvements. Consider removing them for increased readability
- eth-infinitism's `BaseAccount` already inherits from `IAccount`, therefore that inheritance can be removed from `BaseOPF7702` contract.

Recommendation: Consider implementing these suggestions.

Definitions

- **High severity** – High-severity issues usually put a large number of users' sensitive information at risk, or are reasonably likely to lead to catastrophic impact for client's reputation or serious financial implications for client and users.
- **Medium severity** – Medium-severity issues tend to put a subset of users' sensitive information at risk, would be detrimental for the client's reputation if exploited, or are reasonably likely to lead to moderate financial impact.
- **Low severity** – The risk is relatively small and could not be exploited on a recurring basis, or is a risk that the client has indicated is low impact in view of the client's business circumstances.
- **Informational** – The issue does not pose an immediate risk, but is relevant to security best practices or Defence in Depth.
- **Undetermined** – The impact of the issue is uncertain.
- **Fixed** – Adjusted program implementation, requirements or constraints to eliminate the risk.
- **Mitigated** – Implemented actions to minimize the impact or likelihood of the risk.
- **Acknowledged** – The issue remains in the code but is a result of an intentional business or design decision. As such, it is supposed to be addressed outside the programmatic means, such as: 1) comments, documentation, README, FAQ; 2) business processes; 3) analyses showing that the issue shall have no negative consequences in practice (e.g., gas analysis, deployment settings).

Test Suite Results

Test data output was obtained with `make test-all`. The verbose output was removed from the report. All 10 test suites ran with 50 tests executed successfully.

```
Ran 1 test for test/data/Slot.t.sol:Slot
[PASS] test_PrintSlot() (gas: 3318)
Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 910.05ms (289.91ms CPU time)

Ran 1 test for test/proxy/ProxyTest.t.sol:ProxyTest
[PASS] test_AfterInit() (gas: 51107)
Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 3.17s (3.55ms CPU time)

Ran 2 tests for test/proxy/UpgradeImpl.t.sol:UpgradeImpl
[PASS] test_AfterInit() (gas: 51129)
[PASS] test_Upgrade() (gas: 5013764)
```

```
Suite result: ok. 2 passed; 0 failed; 0 skipped; finished in 3.23s (63.95ms CPU time)

Ran 1 test for test/unit/P256.t.sol:P256Test
[PASS] test_ExecuteBatchSKP256() (gas: 581085)
Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 4.06s (779.65ms CPU time)

Ran 8 tests for test/unit/DepositAndTransferETH.t.sol:DepositAndTransferETH
[PASS] test_DepositEthFromEOA() (gas: 28733)
[PASS] test_ExecuteBatchOwnerCall() (gas: 162534)
[PASS] test_ExecuteBatchSKEOA() (gas: 170237)
[PASS] test_ExecuteBatchSKP256() (gas: 722794)
[PASS] test_ExecuteBatchSKP256NonKey() (gas: 717955)
[PASS] test_ExecuteMasterKey() (gas: 1427840)
[PASS] test_ExecuteOwnerCall() (gas: 153559)
[PASS] test_TransferFromAccount() (gas: 24509)
Suite result: ok. 8 passed; 0 failed; 0 skipped; finished in 4.64s (586.77ms CPU time)

Ran 5 tests for test/unit/UpgradeAddresses.t.sol:DepositAndTransferETH
[PASS] test_Addresses() (gas: 26653)
[PASS] test_UpgradeEntryPointAndSendTXWithMasterKey() (gas: 1486473)
[PASS] test_UpgradeEntryPointWithMasterKey() (gas: 1407140)
[PASS] test_UpgradeEntryPointWithRootKey() (gas: 136855)
[PASS] test_UpgradeWebAuthnVerifiertWithRootKey() (gas: 139371)
Suite result: ok. 5 passed; 0 failed; 0 skipped; finished in 4.70s (336.24ms CPU time)

Ran 4 tests for test/unit/Registartion.t.sol:RegistartionTest
[PASS] test_RegisterKeyEOAWithMK() (gas: 1407006)
[PASS] test_RegisterKeyP256NonKeyWithMK() (gas: 1409228)
[PASS] test_RegisterKeyP256WithMK() (gas: 1422777)
[PASS] test_getKeyById_zero() (gas: 47711)
Suite result: ok. 4 passed; 0 failed; 0 skipped; finished in 4.87s (157.35ms CPU time)

Ran 15 tests for test/unit/Execution.t.sol:Execution7821
[PASS] test_ExecuteBatchMasterKey7821() (gas: 1463718)
[PASS] test_ExecuteBatchOfBatches7821() (gas: 262213)
[PASS] test_ExecuteBatchOfBatches7821Reverts() (gas: 153454)
[PASS] test_ExecuteBatchOfBatchesMasterKey7821() (gas: 1599175)
[PASS] test_ExecuteBatchOfBatchesP2567821() (gas: 853139)
[PASS] test_ExecuteBatchOfBatchesP256NonKey7821() (gas: 845337)
[PASS] test_ExecuteBatchOfBatchesSKEOA7821() (gas: 306150)
[PASS] test_ExecuteBatchOwnerCall7821() (gas: 181507)
[PASS] test_ExecuteBatchP2567821() (gas: 756106)
[PASS] test_ExecuteBatchP256NonKey7821() (gas: 757677)
[PASS] test_ExecuteBatchSKEOA7821() (gas: 211422)
[PASS] test_ExecuteBatchSKEOA7821ApproveSendETH() (gas: 233680)
[PASS] test_ExecuteOwnerCall7821() (gas: 150557)
[PASS] test_ExecuteSKEOA7821() (gas: 178809)
[PASS] test_getKeyById_zero() (gas: 35579)
Suite result: ok. 15 passed; 0 failed; 0 skipped; finished in 6.65s (536.74ms CPU time)

Ran 2 tests for test/unit/Keys.t.sol:KeysTest
[PASS] test_RevokeALL() (gas: 1375833)
[PASS] test_RevokeByID() (gas: 108964)
Suite result: ok. 2 passed; 0 failed; 0 skipped; finished in 32.30s (303.24ms CPU time)

Ran 11 tests for test/unit/Recoverable.t.sol:Recoverable
[PASS] test_AfterCancellation() (gas: 68012)
[PASS] test_AfterConfirmation() (gas: 204753)
[PASS] test_AfterConstructor() (gas: 29971)
[PASS] test_AfterProposal() (gas: 42433)
[PASS] test_AfterRevokeConfirmation() (gas: 210463)
[PASS] test_CancelGuardianRevocation() (gas: 242408)
[PASS] test_CancelRecovery() (gas: 257041)
[PASS] test_CompleteRecoveryToEOA() (gas: 351245)
[PASS] test_CompleteRecoveryToWebAuthn() (gas: 479607)
[PASS] test_RevokeGuardian() (gas: 232465)
[PASS] test_StartRecovery() (gas: 305343)
Suite result: ok. 11 passed; 0 failed; 0 skipped; finished in 32.81s (2.39s CPU time)

Ran 10 test suites in 33.01s (97.33s CPU time): 50 tests passed, 0 failed, 0 skipped (50 total tests)
```

Code Coverage

Overall test coverage is at 55% for lines and 26% for branches, leaving room for improvement. While core contracts such as `KeysManager.sol` and `OPF7702.sol` have relatively high line coverage, branch coverage remains low, meaning important conditional paths may be untested.

`BaseOPF7702.sol` and `OPF7702Recoverable.sol` could benefit from more thorough testing of branching logic and edge cases.

File	% Lines	% Statements	% Branches	% Funcs
script/DeployUpgradeable.s.sol	0.00% (0/6)	0.00% (0/7)	100.00% (0/0)	0.00% (0/1)
script/InitProxy.s.sol	0.00% (0/38)	0.00% (0/46)	100.00% (0/0)	0.00% (0/2)
src/core/BaseOPF7702.sol	65.71% (23/35)	53.85% (21/39)	50.00% (2/4)	80.00% (8/10)
src/core/Execution.sol	72.73% (32/44)	75.00% (42/56)	60.00% (6/10)	83.33% (5/6)
src/core/KeysManager.sol	83.33% (70/84)	83.75% (67/80)	50.00% (3/6)	91.67% (11/12)
src/core/OPF7702.sol	79.10% (140/177)	84.69% (177/209)	52.38% (22/42)	100.00% (16/16)
src/core/OPF7702Recoverable.sol	80.30% (163/203)	74.55% (164/220)	4.55% (2/44)	100.00% (24/24)
src/core/OPFMain.sol	66.67% (2/3)	50.00% (1/2)	100.00% (0/0)	100.00% (1/1)
src/libs/Base64.sol	0.00% (0/61)	0.00% (0/57)	0.00% (0/5)	0.00% (0/4)
src/libs/KeyDataValidationLib.sol	100.00% (20/20)	100.00% (30/30)	100.00% (1/1)	100.00% (8/8)
src/libs/KeyHashLib.sol	100.00% (8/8)	100.00% (5/5)	100.00% (2/2)	100.00% (3/3)
src/libs/P256.sol	0.00% (0/40)	0.00% (0/35)	0.00% (0/4)	0.00% (0/5)
src/libs/UpgradeAddress.sol	73.68% (28/38)	72.22% (26/36)	0.00% (0/7)	100.00% (7/7)
src/libs/ValidationLib.sol	88.89% (8/9)	60.00% (6/10)	0.00% (0/4)	100.00% (4/4)
src/libs/WebAuthn.sol	0.00% (0/115)	0.00% (0/115)	0.00% (0/13)	0.00% (0/10)
src/mocks/MockERC20.sol	0.00% (0/2)	0.00% (0/1)	100.00% (0/0)	0.00% (0/1)
src/mocks/SimpleContract.sol	0.00% (0/12)	0.00% (0/10)	0.00% (0/4)	0.00% (0/4)
src/utils/ERC7201.sol	0.00% (0/2)	0.00% (0/1)	100.00% (0/0)	0.00% (0/1)
src/utils/SpendLimit.sol	0.00% (0/8)	0.00% (0/8)	0.00% (0/1)	0.00% (0/1)
src/utils/WebAuthnVerifier.sol	0.00% (0/18)	0.00% (0/17)	100.00% (0/0)	0.00% (0/5)
src/utils/WebAuthnVerifierV2.sol	0.00% (0/14)	0.00% (0/13)	100.00% (0/0)	0.00% (0/3)
test/Base.sol	86.36% (19/22)	88.24% (15/17)	100.00% (0/0)	83.33% (5/6)

File	% Lines	% Statements	% Branches	% Funcs
Total	53.49% (513/959)	54.64% (554/1014)	25.85% (38/147)	68.66% (92/134)

Changelog

- 2025-08-12 - Initial report

About Quantstamp

Quantstamp is a global leader in blockchain security. Founded in 2017, Quantstamp’s mission is to securely onboard the next billion users to Web3 through its best-in-class Web3 security products and services.

Quantstamp’s team consists of cybersecurity experts hailing from globally recognized organizations including Microsoft, AWS, BMW, Meta, and the Ethereum Foundation. Quantstamp engineers hold PhDs or advanced computer science degrees, with decades of combined experience in formal verification, static analysis, blockchain audits, penetration testing, and original leading-edge research.

To date, Quantstamp has performed more than 500 audits and secured over $200 billion in digital asset risk from hackers. Quantstamp has worked with a diverse range of customers, including startups, category leaders and financial institutions. Brands that Quantstamp has worked with include Ethereum 2.0, Binance, Visa, PayPal, Polygon, Avalanche, Curve, Solana, Compound, Lido, MakerDAO, Arbitrum, OpenSea and the World Economic Forum.

Quantstamp’s collaborations and partnerships showcase our commitment to world-class research, development and security. We're honored to work with some of the top names in the industry and proud to secure the future of web3.

Notable Collaborations & Customers:

- Blockchains: Ethereum 2.0, Near, Flow, Avalanche, Solana, Cardano, Binance Smart Chain, Hedera Hashgraph, Tezos
- DeFi: Curve, Compound, Maker, Lido, Polygon, Arbitrum, SushiSwap
- NFT: OpenSea, Parallel, Dapper Labs, Decentraland, Sandbox, Axie Infinity, Illuvium, NBA Top Shot, Zora
- Academic institutions: National University of Singapore, MIT

Timeliness of content

The content contained in the report is current as of the date appearing on the report and is subject to change without notice, unless indicated otherwise by Quantstamp; however, Quantstamp does not guarantee or warrant the accuracy, timeliness, or completeness of any report you access using the internet or other means, and assumes no obligation to update any information following publication or other making available of the report to you by Quantstamp.

Notice of confidentiality

This report, including the content, data, and underlying methodologies, are subject to the confidentiality and feedback provisions in your agreement with Quantstamp. These materials are not to be disclosed, extracted, copied, or distributed except to the extent expressly authorized by Quantstamp.

Links to other websites

You may, through hypertext or other computer links, gain access to web sites operated by persons other than Quantstamp. Such hyperlinks are provided for your reference and convenience only, and are the exclusive responsibility of such web sites' owners. You agree that Quantstamp are not responsible for the content or operation of such web sites, and that Quantstamp shall have no liability to you or any other person or entity for the use of third-party web sites. Except as described below, a hyperlink from this web site to another web site does not imply or mean that Quantstamp endorses the content on that web site or the operator or operations of that site. You are solely responsible for determining the extent to which you may use any content at any other web sites to which you link from the report. Quantstamp assumes no responsibility for the use of third-party software on any website and shall have no liability whatsoever to any person or entity for the accuracy or completeness of any output generated by such software.

Disclaimer

The review and this report are provided on an as-is, where-is, and as-available basis. To the fullest extent permitted by law, Quantstamp disclaims all warranties, expressed implied, in connection with this report, its content, and the related services and products and your use thereof, including, without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement. You agree that access and/or use of the report and other results of the review, including but not limited to any associated services, products, protocols, platforms, content, and materials, will be at your sole risk. FOR AVOIDANCE OF DOUBT, THE REPORT, ITS CONTENT, ACCESS, AND/OR USAGE THEREOF, INCLUDING ANY ASSOCIATED SERVICES OR MATERIALS, SHALL NOT BE CONSIDERED OR RELIED UPON AS ANY FORM OF FINANCIAL, INVESTMENT, TAX, LEGAL, REGULATORY, OR OTHER ADVICE. This report is based on the scope of materials and documentation provided for a limited review at the time provided. You acknowledge that Blockchain technology remains under development and is subject to unknown risks and flaws and, as such, the report may not be complete or inclusive of all vulnerabilities. The review is limited to the materials identified in the report and does not extend to the compiler layer, or any other areas beyond the programming language, or programming aspects that could present security risks. The report does not indicate the endorsement by Quantstamp of any particular project or team, nor guarantee its security, and may not be represented as such. No third party is entitled to rely on the report in any way, including for the purpose of making any decisions to buy or sell a product, service or any other asset. Quantstamp does not warrant, endorse, guarantee, or assume responsibility for any product or service advertised or offered by a third party, or any open source or third-party software, code, libraries, materials, or information to, called by, referenced by or accessible through the report, its content, or any related services and products, any hyperlinked websites, or any other websites or mobile applications, and we will not be a party to or in any way be responsible for monitoring any transaction between you and any third party. As with the purchase or use of a product or service through any medium or in any environment, you should use your best judgment and exercise caution where appropriate.

Quantstamp