Coinbase Session Keys Security Review

Auditors

Noah Marconi, Lead Security Researcher

Riley Holterhus, Lead Security Researcher

Cccz, Security Researcher

Chinmay Farkya, Associate Security Researcher

Report prepared by: Lucas Goiriz

October 1, 2024

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

Base is a secure and low-cost Ethereum layer-2 solution built to scale the userbase on-chain.

Disclaimer: This security review does not guarantee against a hack. It is a snapshot in time of Coinbase Session Keys 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 4 days in total, Coinbase engaged with Spearbit to review the session-keys protocol. In this period of time a total of **15** issues were found.

Summary

Project Name	Coinbase
Repository	session-keys
Commit	bdec9a20
Type of Project	DeFi, Account Abstraction
Audit Timeline	Sep 9th to Sep 13th
Two week fix period	Sep 14th

Issues Found

Severity	Count	Fixed	Acknowledged
Critical Risk	0	0	0
High Risk	1	1	0
Medium Risk	0	0	0
Low Risk	5	1	4
Gas Optimizations	2	1	1
Informational	7	7	0
Total	**15**	**10**	**5**

5 Findings

5.1 High Risk

5.1.1 Permissions can drain approvals given to certain paymasters

Severity: High Risk

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

Description: The Smart Wallet Permissions system requires all user operations utilize an allowlisted paymaster. This is because without this requirement, a permission could arbitrarily use the Smart Wallet's ETH for gas fees, which would increase the trust needed for each permission.

Currently, paymaster allowlisting is enforced via the `isPaymasterEnabled` mapping, which is checked in the `beforeCalls()` function:

```
function beforeCalls(Permission calldata permission, address paymaster, address userOpCosigner)
    external
    whenNotPaused
{
    //...
    if (!isPaymasterEnabled[paymaster]) revert DisabledPaymaster(paymaster);
    // ...
}
```

Note that the `beforeCalls()` function is invoked during the execution phase of the user operation, which implies a non-allowlisted paymaster can successfully pass the `validateUserOp()` step of the ERC-4337 transaction. This poses a problem, because some paymasters are authorized to debit the account as long as the validation phase succeeds.

For example, some ERC20 paymasters are designed to withdraw ERC20 tokens from the account in exchange for covering transaction fees. These paymasters will always charge the account if the validation phase succeeds, regardless of whether the execution phase is successful.

As a result, if an account has pre-approved a paymaster outside of the Smart Wallet Permissions system, any of its permissions could potentially drain the pre-approved balance. This would be achieved by using the paymaster during the validation phase, even though it will fail the `beforeCalls()` step later. Since the paymaster would still charge the user, and since the permission could set excessively high gas costs, the account's balance that it has pre-approved could be drained quickly.

Note that this behavior is not preventable by the cosigner, since the cosigner check is also in the execution phase.

Recommendation: Consider moving the paymaster allowlist check into the validation phase. Since the `isPaymasterEnabled` mapping would violate the ERC-4337 storage rules during the validation step, this change would require a refactor of the paymaster check. One possible approach is to include the allowed paymaster as a field in the `Permission` struct.

Coinbase: One possible fix for the core issue has been implemented in PR 53. Also, an additional safeguard for ensuring the paymaster is non-zero in `isValidSignature()` has been added in PR 54.

Spearbit: Verified.

The fix in PR 53 has changed the `cosigner` address into an immutable variable, allowing the cosigner check to be moved into the validation stage. As a result, as long as the `cosigner` correctly only signs user operations with approved paymasters, the validation step will not pass with any unauthorized paymasters. This resolves the core issue. After a discussion with the Coinbase team, this fix was verified as a potential solution but may not be the final change chosen. Coinbase is exploring other options that may involve larger refactors and potential re-audits.

The fix in PR 54 will help prevent similar issues in future permission contracts that don't enforce a paymaster.

5.2 Low Risk

5.2.1 No user-control on what selector is called on an external contract allowed under a session key

Severity: Low Risk

Context: PermissionCallableAllowedContractNativeTokenRecurringAllowance.sol#L126-L137

Description: The `validatePermission()` function only checks that the primary selector is `permissionedCall()` (as the overall call is wrapped with this selector before sending the userOp), but there are no user-level controls on what selector is actually going to be called on the `allowedContract`.

The external contract having `permissionedCall` is a system-wide requirement (for the current permission contract) but the user has no say in what function gets called inside the `permissionedCall()` [ie. the self-delegatecall part].

So, a session key might convince a smart account to sign a permission to call `allowedContract` by telling them that they are going to call only function A, but there can be a problem if there are multiple functions in the `allowedContract` that have different capabilities.

For example, if the user believes the session key will only be allowed to stake ETH, but it actually sends a transaction encoding a call to lend ETH, the call will go through. If the user is not aware of this behavior, they may be surprised by this.

This is possible if the `allowedContract` supports multiple selectors under `permissionedCall()`.

Recommendation: Consider adding user-control on the "actual selector" called inside the `permissionedCall()` (by including it in the `permissionHash`) so that only one selector approved by the user is accessible under a session key.

Coinbase: Acknowledged. The current behavior is intended, as users are expected to trust permissions they have authorized, including all selectors accessible under the permission.

Spearbit: Acknowledged.

5.2.2 `MagicSpend.withdraw()` calls are exposed to frontrun attacks

Severity: Low Risk

Context: PermissionCallableAllowedContractNativeTokenRecurringAllowance.sol#L130-L140

Description: The protocol supports the `magicSpend.withdraw()` permission call.

```
} else if (selector == MagicSpend.withdraw.selector) {
    // check call target is MagicSpend
    if (call.target != magicSpend) revert CallErrors.TargetNotAllowed(call.target);

    // parse MagicSpend withdraw request
    MagicSpend.WithdrawRequest memory withdraw =
        abi.decode(BytesLib.trimSelector(calls[i].data), (MagicSpend.WithdrawRequest));

    // check withdraw is native token
    if (withdraw.asset != address(0)) revert InvalidWithdrawAsset(withdraw.asset);
```

`magicSpend.withdraw()` call needs to include signature in the `WithdrawRequest` param and the signature will be consumed.

For a given signature, as long as the call is sent from the same smart wallet, the call is valid.

```
function withdraw(WithdrawRequest memory withdrawRequest) external {
    if (block.timestamp > withdrawRequest.expiry) {
        revert Expired();
    }

    if (!isValidWithdrawSignature(msg.sender, withdrawRequest)) {
        revert InvalidSignature();
    }

    _validateRequest(msg.sender, withdrawRequest);
// ...
function _validateRequest(address account, WithdrawRequest memory withdrawRequest) internal {
    if (_nonceUsed[withdrawRequest.nonce][account]) {
        revert InvalidNonce(withdrawRequest.nonce);
    }

    uint256 maxAllowed = address(this).balance / maxWithdrawDenominator;
    if (withdrawRequest.asset == address(0) && withdrawRequest.amount > maxAllowed) {
        revert WithdrawTooLarge(withdrawRequest.amount, maxAllowed);
    }

    _nonceUsed[withdrawRequest.nonce][account] = true;
```

When the user approves Apps, App's permission calls are sent from that user's smart wallet, and when one App's permission calls include `MagicSpend.withdraw()`, the signature in the `WithdrawRequest` param is public, which give other App's permission calls the opportunity to include that `MagicSpend.withdraw()` call.

Consider the following case:

1. The user approves App1 and App2, App2 does not spend any funds, App1 needs some funds initially, so the user provides a `MagicSpend` signature to allow App1 to withdraw the funds.
2. App1 constructs the user operation A, which wants to call `magicSpend.withdraw()` to withdraw funds and perform subsequent spend calls.
3. The misbehaving App2 constructs the user operation B, which includes the `magicSpend.withdraw()` call with the signature from user operation A.
4. User operation B is packaged by the Bundlers earlier, so that user operation A fails and the withdrawn funds remain in the smart wallet instead of being spent.
5. And worse, if App2 is approved to spend funds but has no funds in the smart wallet, App2 may spend those withdrawn funds instead of App1.

Recommendation: After discussing with the sponsor, it would be good to use cosigning to mitigate. It will cache the most recent magic spend withdraws and rejects duplicates.

And may also need similar mitigation for some unique permissionedCalls (e.g. containing signatures) in the future.

Coinbase: Acknowledged.

Spearbit: Acknowledged.

5.2.3 `bytes4` casting can be unsafe

Severity: Low Risk

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

Description: There are three locations in the codebase where `bytes` values representing calldata are typecast to `bytes4` to determine the function selector of a call. These locations are:

1. The `isValidSignature()` function within the `PermissionManager` contract:

```
if (bytes4(data.userOp.callData) != CoinbaseSmartWallet.executeBatch.selector) {
    revert CallErrors.SelectorNotAllowed(bytes4(data.userOp.callData));
}
```

2. The `validatePermission()` function within the `PermissionCallableAllowedContractNativeTokenRecurringAllowance` contract:

```
bytes4 selector = bytes4(call.data);
if (selector == IPermissionCallable.permissionedCall.selector) {
    // ...
} else if (selector == MagicSpend.withdraw.selector) {
    // ...
} else {
    revert CallErrors.SelectorNotAllowed(selector);
}
```

3. The `permissionedCall()` function within the `PermissionCallable` contract:

```
if (call.length < 4) revert InvalidCallLength();
if (!supportsPermissionedCallSelector(bytes4(call))) revert NotPermissionCallable(bytes4(call));
```

In general, note that the conversion of a `bytes` value shorter than 4 bytes into a `bytes4` value will not revert, and will instead add extra zero bytes as padding. For example, consider the following test case:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.23;

import {Test} from "forge-std/Test.sol";

contract Bytes4Casting is Test {
    function test_bytes4_casting() public pure {
        bytes memory data = hex"112233";
        bytes4 sig = 0x11223300;

        assertEq(data.length, 3);
        assertEq(bytes4(data), sig);
    }
}
```

This behavior is not a concern in location (3) above, due to the explicit length check. In location (1), the `bytes` are later decoded in a way that will revert if shorter than 4 bytes, which also mitigates any risk.

However, in location (2), there is technically nothing preventing the `bytes` value from being less than 4 bytes in length. If either `IPermissionCallable.permissionedCall.selector` or `MagicSpend.withdraw.selector` had trailing zero bytes, it's possible that a shortened `bytes` value could be implicitly padded and would match the overall selector. This would introduce other concerns since when the call executes later on, it would reach the contract's fallback function instead of the function that was expected.

Fortunately, `IPermissionCallable.permissionedCall.selector` is `0x2bd1b86d` and `MagicSpend.withdraw.selector` is `0xd833cae`, so there are no trailing zeroes and this is not an issue. However, to be safer, this behavior could be eliminated entirely

Recommendation: To mitigate any potential problems in the future, consider adding checks in locations (1) and (2) to ensure the `bytes` values are at least 4 bytes long.

Coinbase: Addressed in PR 52.

Spearbit: Verified. The check has been added in location (2). Location (1) has been left as is, which is safe because short calldata will eventually trigger a revert regardless.

5.2.4 Cosigner signatures are not revocable

Severity: Low Risk

Context: PermissionManager.sol#L200-L203

Description: Cosigner signatures are not revocable. In the event of a delay before a transaction is included in a block, and new information surfaces such as a contract upgrade, a revocation may be desirable.

Recommendation: Consider adding in an expiry or nonce to allow for explicit revocation.

Coinbase: Acknowledged.

Spearbit: Acknowledged.

5.2.5 Cosigner event exclusions are limited

Severity: Low Risk

Context: PermissionManager.sol#L343

Description: The cosigner signing is handed by a backend service monitoring for unintended transactions that may move value away from a Smart Wallet account. In the project briefing and documentation, Coinbase notes that simulations will look for logs matching `ERC20/ERC721/ERC1155 transfer logs where the from argument is the user's address (userOp.sender)`; similarly for approvals.

There may be value transfer events not covered with the cosigner relying on `Transfer/Approve` events for its veto. Examples include an orderbook already approved by the Smart Wallet, session keys can create unfavorable orders. Other admin functions like `Ownable2Step` won't emit `Transfer/Approve` events but are not likely to be desirable calls from a permissioned signer.

Recommendation: Broadly, encouraging users to exercise caution in choosing applications to approve permissions for. A vetted registry of known safe applications may assist.

Considering additional events such as:

- Order creation / modification.
- Direct interactions with Univ4 Core (circumventing the LP token transfer events).
- Ownership transfers.
- etc...

External to individual `userOps`, monitoring for contract upgrade events will also be important. See the issue "Cosigner signatures are not revocable" for revocation related recommendation well suited for handling contract upgrades.

Coinbase: Acknowledged.

Spearbit: Acknowledged.

5.3 Gas Optimization

5.3.1 Save gas by changing checks for previously saved `RecurringAllowance` in `_initializeRecurringAllowance()`

Severity: Gas Optimization

Context: NativeTokenRecurringAllowance.sol#L123

Description: Some gas can be saved by changing the condition in if clause to either `! recurringAllowance.start > 0` or `! recurringAllowance.period > 0`.

Both do not need to be checked as it is guaranteed that if the recurring allowance has been saved previously, both these values would be non-zero.

Coinbase: Acknowledged.

Spearbit: Acknowledged.

5.3.2 Save gas in reverting transactions by moving hashing until after checks

Severity: Gas Optimization

Context: PermissionManager.sol#L322

Description: `permissionHash` is not referenced until the `IPermissionContract.validatePermission` call. In the reverting cases, the gas to hash `data.permission` can be saved by moving the operation to later in the function.

Coinbase: Addressed in PR 51.

Spearbit: Verified.

5.4 Informational

5.4.1 `lastCycleExists` can be simplified

Severity: Informational

Context: NativeTokenRecurringAllowance.sol#L203

Description: Checking that `start != 0` is enough to determine if the last cycle existed. In case that start > 0, the end will automatically be initialized as a non-zero value(using start value) as seen in the else branch here.

If `spend == 0` or `spend > 0` also does not impact the logic of determining if a last cycle existed. This is because the cycle is only cached in storage if the spend value is non-zero.

Recommendation: Change it to `bool lastCycleExists = lastCycleUsage.start != 0;`.

Coinbase: Fixed in PR 51.

Spearbit: Verified.

5.4.2 Documentation errors

Severity: Informational

Context: NativeTokenRecurringAllowance.sol#L181, NativeTokenRecurringAllowance.sol#L57, PermissionCallableAllowedContractNativeTokenRecurringAllowance.sol#L96, PermissionManager.sol#L243-L246, PermissionManager.sol#L267-L271, PermissionManager.sol#L315

Description: The code comments are wrong at some places:

1. Wrong comment for event `RecurringAllowanceInitialized()`. Should be ⇒ `Register native token allowance for a permission.`
2. Wrong comment for `_getCurrentCycleUsage()` function. Should be ⇒ `n*recurringAllowance.period - 1`.
3. Wrong comment for `validatePermission()` function. Should be ⇒ `Offchain userOp construction should append useRecurringAllowance call to calls array.`
4. Wrong comment for `revokePermission()` function. Remove it.
5. Confusing comment for `setPaymasterEnabled()` function. Remove it as the intention is to ban no-paymaster ops at the manager level. For more consistency, also add a check that `address(0)` can't be enabled as a paymaster here.
6. Wrong comment for `isValidSignature()` function. Should be ⇒ `Verifies that userOp.calldata calls CoinbaseSmartWallet.executeBatch.`

Recommendation: Correct the documentation as suggested

Coinbase: Addressed comment changes in PR 51. Also, point (5) relates to PR 54.

Spearbit: Verified.

5.4.3 `getRequiredPrefund()` can be removed

Severity: Informational

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

Description: The `UserOperationLib` implements the `getRequiredPrefund()` function, however this function is not used in the current version of the codebase.

Recommendation: Consider removing the `getRequiredPrefund()` function.

Coinbase: Fixed in PR 51.

Spearbit: Verified.

5.4.4 `beforeCalls()` and `useRecurringAllowance()` payable considerations

Severity: Informational

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

Description: The `beforeCalls()` and `useRecurringAllowance()` functions must be called in a specific manner during the execution phase, which is enforced by checks in the validation phase. The `beforeCalls()` check is implemented as follows:

```
// check first call is valid `self.beforeCalls`
if (calls[0].target != address(this) || !BytesLib.eq(calls[0].data, beforeCallsData)) {
    revert InvalidBeforeCallsCall();
}
```

The `useRecurringAllowance()` check is implemented similarly:

```
// check last call is valid `this.useRecurringAllowance`
CoinbaseSmartWallet.Call memory lastCall = calls[callsLen - 1];
if (lastCall.target != address(this) || !BytesLib.eq(lastCall.data, useRecurringAllowanceData)) {
    revert InvalidUseRecurringAllowanceCall();
}
```

Notice that neither check enforces that the ETH value of the call is 0. This makes it crucial that neither `beforeCalls()` nor `useRecurringAllowance()` are defined as payable functions. Otherwise, ETH could be transferred by permissions in an untracked manner.

Fortunately, both `beforeCalls()` and `useRecurringAllowance()` are indeed not defined as `payable`, so there is no issue.

Recommendation: To prevent any possible errors in future changes, consider documenting this behavior above the `beforeCalls()` and `useRecurringAllowance()` functions. Alternatively, depending on preferences for gas efficiency, consider adding a redundant check to both calls that their ETH value is 0.

Coinbase: Fixed in PR 51.

Spearbit: Verified. A check has been added in both locations that ensures the call value is 0.

5.4.5 `EIP1271_MAGIC_VALUE` visibility not explicitly set

Severity: Informational

Context: PermissionManager.sol#L53

Description: Silence linter warning by specifying visibility.

Coinbase: Fixed in PR 51.

Spearbit: Verified.

5.4.6 Permissions cannot be un-revoked and must be recreated with modified `Permission`

Severity: Informational

Context: PermissionManager.sol#L246-L254

Description: Permission revocation cannot be undone.

Not a problem for currently supported permissions as the start time can be changed to re-enable. For future permissions, users attempting to re-enable an identical permission to a previously revoked one will need to select a new expiry timestamp to produce a unique `permissionHash`.

Recommendation: No code change recommended. A note added to documentation would be useful to users attempting to re-enable a permission.

Coinbase: Added note to documentation in PR 51.

Spearbit: Verified.

5.4.7 `CoinbaseSmartWallet` and similar contract accounts cannot be a permission signer

Severity: Informational

Context: PermissionManager.sol#L338-L340

Description: `permission.signer` cannot be a standard `CoinbaseSmartWallet` as its own storage is not keyed by the user address.

No issue identified as simulation will fail and the transaction would be dropped by the bundler.

Recommendation: Add note to documentation.

Coinbase: Added note to documentation in PR 51.

Spearbit: Verified.