Session Keys Internal Audit [Public]

By: Coinbase's Protocol Security Team

Audit Scope

Coinbase's internal audit of Session Keys took place in parallel to the development of the protocol. Several commit hashes and PRs were reviewed during the audit.

The following files were in scope for our internal review:

Permission Manager:
PermissionManager

Permissions:
PermissionCallableAllowedContractNativeTokenRecurringAllowance

Mixins:
NativeTokenRecurringAllowance
PermissionCallable

Utils:
BytesLib
SignatureCheckerLib
UserOperationLib
CallErrors

System Overview

The session keys system is designed in a modular way that enables the Permission Manager contract to contain the core logic checks, while each individual permission contract validates the permission at a more granular level. This design allows different permission contracts to be added / removed, such as a permission to spend Ether allowances or a permission to spend ERC20 token allowances. In order for a permission to be valid and a user operation to be executed, 3 important signatures are needed:

1. The approval signature from a Smart Contract Wallet owner
2. The signature from the session key, which is signed over each user operation
3. The signature form the Coinbase cosigner, which is also signed over each user operation

In order for a permission to be initially approved, the approvePermission function has to be called. This can be validated if the caller is the account that the permission is seeking approval for or if the caller has provided a valid signature from an owner on the smart wallet account.

Approved permissions are then initialized and saved to the storage of the individual Permissions contract. Although permissions are saved in the state of each individual Permission contract, the entrypoint for initializing permissions is only through a call from the Permission Manager contract itself.

For a Dapp to use the approved permission in a user operation, there are 2 signatures needed: a signature from the **session key** signed over the user operation and a signature from the **Coinbase cosigner**, also signed over the user operation. For each user operation that the session key wants to execute, there are a few important safety checks to ensure that the session key cannot arbitrarily move funds from a user's account. These checks include, but are not limited to, ensuring the user operation can only call the contract address specified by the permission, as well as ensuring that only certain function selectors on that allowed contract are allowed.

The Coinbase cosigner is an additional safety feature that must sign every user operation that uses a Permission. The cosigner can ***only veto*** session key user operations, rather than initiate transactions on its own. In this way, it limits the trust assumptions of the cosigner, where it can only act as a vetoing mechanism to reject session key user operations by simply not signing the user operation.

More detailed diagrams on how the Coinbase cosigner receives and signs the user operation can be found in the repo's documentation folder.

Permission Manager

The Permission Manager contract contains the core logic to perform checks in both the validation phase and the execution phase of a user operation's transaction lifecycle. These preliminary checks in the validation phase are applied where EIP-7562 validation constraints are allowed. For example, the expiration of a permission cannot be checked during validation because it needs to be compared against the block timestamp. For these reasons, general purpose checks that can be done within isValidSignature are performed, while the rest of the checks are offloaded to the execution phase of the user operation, primarily in the beforeCalls function.

Validation Phase:

During the validation phase, the Permission Manager will verify a few checks such as:

- The account that the permission is for matches the userOp's sender
- The permission is not revoked
- The permission is approved - this is done either through storage or through a just in time approval with a signature from the smart wallet's owner
- The user operation is signed by the session key
- The userOp is only calling executeBatch on the smart wallet
- The first call in executeBatch calls the beforeCalls function on the Permission Manager contract itself

- No self calls (reentrancy) are allowed on the smart wallet
- No reentrancy calls are allowed on the Permission Manager

Execution Phase:

Execution phase checks are done within the beforeCalls function in the Permission Manager contract. This is done by ensuring (during the validation phase), that the first call in the calldata of the user operation specifically calls the beforeCalls function with the correct arguments. The reason that these checks are performed in the execution phase, rather than the validation phase is because of account abstraction constraints, such as accessing a contract's storage and block information such as block timestamp. In this flow, it allows the user operation to be properly validated, but during the execution phase rather than the validation phase.

Checks during the execution phase include:

- The permission is not expired
- The Permission Manager is not paused
- The specified paymaster is enabled (allowed)
- The user operation is signed by the Coinbase cosigner

NativeTokenReccuringAllowance

In order for a Permission to be used, it first must be approved by a Smart Wallet owner and then initialized on the NativeTokenRecurringAllowance contract. Initialization can only occur once per approved permission and the permissioned allowance is saved in the storage of the individual Permission contract, rather than on the Permission Manager itself. Once initialized, the session key is able to spend the given amount per cycle on a recurring basis, given that it's passed all the validation checks.

Similar to the Permission Manager checks, there are additional validation checks on the individual permission contract to ensure that more granular checks are done. These checks include:

- A paymaster must be used
- Magic Spend cannot be used as a paymaster (but can be used to withdraw funds)
- Only the permissionedCall function selector is allowed to be called on the allowed contract specified by the given permission
- If the call is using Magic Spend, only the withdraw function can be used. Additionally, the withdraw asset must be Ether
- The last call must be useRecurringAllowance to ensure spends are accounted for

Allowances are structured on a per cycle and recurring basis. It is similar to a subscription based model, where a set amount is able to be spent each cycle, which lasts until the permission itself is expired or the permission has been revoked by the user. For example, take the following allowance into consideration:

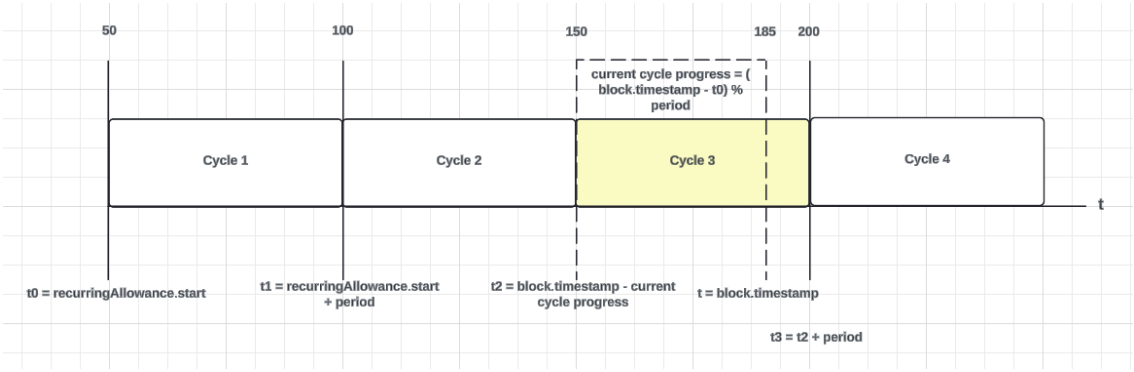
```
Unset
   struct RecurringAllowance {
        /// @dev Start time of the recurring allowance's first cycle (unix
seconds).
        uint48 start;
        /// @dev Time duration for resetting spend on a recurring basis
(seconds).
        uint48 period;
        /// @dev Maximum allowed value to spend within a recurring cycle
        uint160 allowance;
    }

Start = 50
Period = 50
Allowance = 250
```

The first time a session key would be able to use this approved allowance is at time = 50 because that is the start of the first cycle. Because this cycle lasts for a period of 50 seconds, the end time for the first cycle would be at time = 100. This is to say that during the period of time between 50 => time < 100, the session key is allowed to spend 250 of value. Once time progresses to 100, the cycle's allowance resets and for the duration where 100 >= time < 150, the session key is allowed to spend another 250 of value on behalf of the user.

In order to account for a user's spend during a given cycle, the _getCurrentCycleUsage function is used. The goal of this function is to return the current active cycle (highlighted in yellow below) and account for how much of the allowance in the given cycle has already been spent. If the current cycle is still active, then the same active cycle parameters will be returned (with the amount already spent in the cycle). This is desired because any spends in subsequent calls (between start and end period of the current cycle) would add to the total spend of the current cycle and the accounting logic needs to ensure that this total spend does not exceed the given allowance for a cycle.

Anytime an allowance is used, its current usage parameters get saved as the latest cycle for that specific account's permission. When using the same permission again to spend the allowance, the active cycle is determined by if the current time has exceeded the end of the previous cycle or not. If the current time had not exceeded the end period, then that means the last cycle is still active and any spends will be accrued towards the current cycle's total allowance. However, if the spend is past the last active cycle, this means that we are now in a new cycle with a new allowance. Spends get reset to 0 and then any spends in the current call will be added to the 0 spend, signally a fresh cycle. The diagram below demonstrates this example:



Privileged Roles:

Privileged roles are set at the Permission Manager level, rather than at each individual Permission contract. The roles can be broken down into the **owner**, **cosigner** and **pendingCosigner**. While the cosigner and pendingCosigner do not have admin functionality on the smart contract itself, these addresses are important to cosign user operations and are used to validate signatures against.

The **owner** can:

- Pause the Permission Manager contract
- Unpause the Permission Manager contract
- Set / unset allowed individual permission contracts
- Set / unset allowed paymasters
- Set / unset if a paymaster's gas spend should be added to total spend
- Set a pending cosigner
- Remove and rotate a cosigner address
- Set a new owner through a two-step owner rotation

Executive Summary

In total 1 high, 5 mediums, 6 lows, and 6 information findings were discovered throughout the review. The high finding involved incorrectly resetting a cycle's spend amount, while other findings include user operations failing during execution, resulting in users' gas funds being lost. A mix of both onchain (smart contract) and offchain mitigations were put in place to address the following findings.

Findings and Known Issues

High Severity

Title	First cycle can allow infinite spend if recurring allowance starts at 0

ID	H-01
Description	When the session key wants to use an approved permission and spend an allowance, the last call the smart wallet account makes has to be useRecurringAllowance to register that the value spent during the array of Calls in executeBatch has been properly accounted towards the permission's allowance. Given a cycle, when the call to useRecurringAllowance is made, the account can either be in an active cycle, or in a new cycle. Take the following example: Unset `cycle 1 cycle 2 cycle 3` `|----|--|-----|-----------|------------|` `0 4 6 12 24 36` `start = 0` `period = 12` `allowance = 100` `first spend ($50) is at t = 4` `second spend ($25) is at t = 5` The first time that the session key spends its permissioned allowance is at t = 4, which falls within the first active cycle. When useRecurringAllowance is called, it will call _getCurrentCycleUsage to determine how much of the active (or new) cycle has been used. Because this is the first spend of the cycle, there are no registered spends saved in storage for the last cycle, and the current cycle usage returned is 0 as expected. The total spend of the function call is then added to this cycle's usage and saved in storage. For example, if we spent a value of $50 at t = 4, then the cycle's total spend would now be saved as $50. The problem of infinite spends arises on subsequent spends when the recurring allowance starts at t = 0. In the same example, if we spend another $25 at t = 5, the same flow would be processed. useRecurringAllowance would need to be called, which then calls _getCurrentCycleUsage to determine the cycle's spend. Because t = 5 falls within the active cycle (first cycle is between 0 >= time < 12), the previous spend of $50 should have been registered and returned as a result of calling _getCurrentCycleUsage. However, this is not the case. The logic within this function only returns the last cycle's usage if lastCycleUsage.start > 0. As a result, a new cycle gets returned again, which sets the tracked total spends back to 0. The total spend of the function call is then added to the cycle's usage and saved in storage, however the total spend is registered as only $25, even though the session key has already spent $75. As long as the first cycle is active, the process of spending and then resetting the total spend can be

	repeated over and over to drain the user's funds even beyond the approved permission allowance amount.
Recommendation	Consider only allowing allowances to be initialized with a start value greater than 0.
Update	Fixed in PR #32 and #33. For a last cycle to be considered to exist, its cycle usage start, end and spend all must be greater than 0. Additionally, an allowance must be initialized with a start and period that is greater than 0.

Medium Severity

Title	Multiple user operations in the same bundle may pass validation checks, but fail during execution
ID	**M-01**
Description	Given 2 user operations in the same bundle, using a Native Token Recurring Allowance permission, there exists a case where both user operations may pass the validation phase, but the second user operation may fail during execution. For example, a user has a spend limit of $100 ETH during a given time period. The first user operation wants to spend $75 ETH, which passes the validation phase. The second user operation wants to spend $50 ETH, which also passes the validation phase. However, during the execution phase, the first user operation would be executed, decreasing our spend allowance to $25 ETH, causing the second user operation to fail. Other cases such as a permission being valid during the validation phase simulation, but failing during the execution phase also exist (due to not being able to access block.timestamp in the validation phase). The impact of such cases are that users are still required to pay gas costs, while their user operation is not executed, resulting in funds lost.
Comments	We are aware that in general, these cases may exist since we have various checks in both the validation phase and in the execution phase in our smart contracts. On our backend, we can enforce these checks through our cosigner. Two paths to mitigation through our cosigner are: 1. Retroactively blocking apps who are submitting many user operations for the same address in a given bundle 2. Limiting each bundle to 1 or less user operation per address Additional mitigation paths considered (will not implement): 1. Simulation that considers userOp mempool (hard) on our backend before cosigner signs

	2. Ask bundlers to not submit ops failing at execution phase 3. Move timestamp to validation phase to disincentivize bundlers from submitting failing ops, but we need them to accept TIMESTAMP opcode 4. Add timestampAttestation from cosigner into userOp calldata to do time-based allowance checks in validation phase without using TIMESTAMP opcode.
Update	This issue has been fixed in PR #26 - session key transactions are now required to use paymasters and paymasters (the Dapp) will bear the gas cost upfront. If execution of the userOp is successful, users can optionally pay back the paymaster. The insight is to set a principle that permissioned userOps should never spend the users assets upfront and only upon successful execution does the user pay. This aligns incentives better with apps, who will bear the upfront cost of a transaction, while still preserving their ability to get debited by the users balance within execution. If an app is submitting transactions that validate but fail at execution, they will own this cost, eliminating this attack vector on user balances. The implementation of this is simply denying address(0) as an allowed paymaster (hard-coded into PermissionManager) and allowing a native token transfer back to the paymaster address within the permission contract validation. If a paymaster is debited by the user, we'll pick it up in the same loop for all other recurring allowance enforcement logic and the user doesn't incur cost if execution fails. This pattern also follows from how ERC20 paymasters are designed where the paymaster/app is responsible for ensuring execution in order to get paid at the end.
Update 2	The functionality to optionally pay back the paymaster is no longer available. Session keys transactions using the Native Recurring Allowance will still always require paymasters and this fix is in commit *6cb881f030bc05e26f70e72a1be11bb2d069b6f7*.

Title	Paymaster gas spend check violates ERC-4337 storage access validation rules
ID	**M-02**
Description	The call to shouldAddPaymasterGasToTotalSpend is accessing storage during the validation phase and that storage is not keyed by the account as the last key. This violates the validation rules according to the EIP-7562 spec.
Recommendation	Consider moving the paymaster gas spend check to the execution phase. Fixed in PR #7.
Update	This feature has been removed and is no longer available. All session key transactions using the Native Recurring Allowance permission must use a paymaster that is not Magic Spend.

Title	Paymaster refund can be used to drain a user’s allowance
ID	**M-03**

Description	When a user operation is constructed, the callGasLimit, verificaitonGasLimit, and preVerificaitonGas of the user operation go into the calculation of the required prefund amount that is withheld by the entrypoint to cover the gas fee for the bundler. The current implementation requires that each session key transaction requires a paymaster to be used. As such, failed user operations during execution will offload the required gas cost to the paymaster, rather than the user. Users would then pay back the paymaster in one of their calls to refund this paymaster. However, there is a griefing vector in the way that user operations may be constructed. A user operation can be constructed in a way such that the required prefund gas amount is extremely high (up to the user's permissioned spend allowance). In this situation, a user operation would be executed with the paymaster fronting the high gas cost, and then having the user pay back this gas cost in one of the calls. Although the paymaster does not gain any additional funds from this attack vector, it could cause the user's funds to be drained, up to the permissioned spend allowance.
Recommendation	The impact of this attack vector can be mitigated by the Coinbase cosigner. Because Coinbase has the ability to construct the gas values within each user operation, as well as the ability to veto (not sign) user operations, consider adding checks to ensure that the constructed user operation falls within a reasonable gas limit. Alternatively, this check can also be performed onchain by declaring an immutable value set as the required prefund limit. However with this approach, its value would not be able to be changed and the justification for such a value could be arbitrary.
Update	This feature has been removed and is no longer available. During the execution of the Calls, users no longer have the option to pay back the paymaster.

Title	Session keys can abuse permission spends while performing no-ops
ID	**M-04**
Description	Permissions are limited by a specified recurring allowance amount, as well as a specific allowed contract address and function selector. The allowed contract address is specified upon permission approval and the only allowed function selector on that contract address to be called is the permissionedCall selector. While this function selector is enforced, its execution logic inside of this function is not. For example, given an approved permission on an allowed contract that is supposed to mint an NFT, it's permissionedCall logic could be structured in a way that does not perform the desired action (minting an NFT), but does spend value:

	Unset `function permissionedCall(bytes calldata call) external payable returns (bytes memory res) {` `  // spend value but do nothing` `  return "0xDoNothing"` `}` This is an abuse vector of permissioned contracts that could result in malicious session keys draining up the given allowance of a user's permission.
Recommendation	Considering that users need to explicitly sign permission approvals, users should be made aware as to what contracts they are signing approvals for. From Coinbase's side, retroactive actions can be performed to specifically blocklist permissed contracts that are acting maliciously to perform no-ops, while spending user's funds. Coinbase does not have the authority to blocklist permissions from being approved, but we can internally keep track of what permissioned contracts are unsafe and make users aware.

Title	Permissioned calls with multiple user operations in the same bundle can be abused to intentionally fail the latter user operations during execution
ID	**M-05**
Description	A similar abuse vector as issue M-04, combined with issue M-01 exists. If there are 2 user operations in the same bundle, combined with a contract that implements malicious permissionedCall logic, users could lose funds due to validation phase checks passing, but the user operation failing during execution time. An example of the attack can be demonstrated as followed: lastBlock = 0 Block.number = 100 Unset `function permissionedCall(bytes calldata call) external payable returns (bytes memory res) {` `    Attack.failTwoUserOpsInSameBlock();` `}` `contract Attack {` `  uint256 public lastBlock;` `  function failTwoUserOpsInSameBlock() {` `  require(block.number != lastBlock);`

| | ```
            lastBlock = block.number;
        }
    }
```<br><br>In this bundle, the first time that failTwoUserOpsInSameBlock() is called by the first user operation, it would set the lastBlock = 100. Then, in the second user operation (in the same bundle), when failTwoUserOpsInSameBlock() is called again, it would fail because the require(block.number != lastBlock) check does not pass, where block.number = 100 and lastBlock = 100. The account who operates the second user operation would still need to pay for gas (if no paymaster was specified), while their user operation failed, causing funds to be drained from the user's smart wallet account. |
| **Recommendation** | This issue has been fixed in PR #26 - session key transactions are now required to use paymasters and paymasters (the Dapp) will bear the gas cost upfront. If execution of the user op is successful, users can optionally pay back the paymaster. |
| **Update** | The functionality to optionally pay back the paymaster is no longer available. Session keys transactions using the Native Recurring Allowance will still always require paymasters and this fix is in commit *6cb881f030bc05e26f70e72a1be11bb2d069b6f7*. |

Low Severity

Title	Solday's ECDSA does not protect against signature malleability
ID	**L-01**
| **Description** | During the validation phase, the Coinbase cosigner address is recovered using Solday's ECDSA library, which is prone to signature malleability. However, the Coinbase cosigner is signing over the hash of the user operation which contains a unique nonce:<br><br>Unset<br>```
struct UserOperation {
    address sender;
    uint256 nonce;
    bytes initCode;
    bytes callData;
    uint256 callGasLimit;
    uint256 verificationGasLimit;
    uint256 preVerificationGas;
    uint256 maxFeePerGas;
    uint256 maxPriorityFeePerGas;
``` |

	`` bytes paymasterAndData; } ``` Because of this, even if the signature is malleable, the user operation would not be able to be replayed, given the entry point will invalidate a used nonce, as signature malleability implies that the signature is used on the same exact data that it was previously signed on.
Recommendation	We are aware of this issue and do not see any concerns with signature malleability for our use case. From a security perspective, there is no impact for signature malleability here.

Title	Setting pending cosigner lacks address(0) check
ID	**L-02**
Description	When setting a pending cosigner, there is not a check to see if the address of the new cosigner is address(0). Impact is if a pending cosigner is set to address(0) and then the cosigner is rotated, the cosigner would be unable to rotate since it reverts if the pendingCosigner == address(0)
Recommendation	Consider checking the newCosigner != address(0) when setting a pending cosigner. This is also documented in the slither finding.
Update	Fixed in PR #22. The address(0) check has been added when setting a pending cosigner, as well as adding an additional resetPendingCosigner function to set the pending cosigner to address(0).

Title	Signature validation reverts on approved signatures
ID	**L-03**
Description	Signature validation reverts on approved and valid signatures and does not revert on invalid signatures and approvals.
Recommendation	Consider flipping the revert cases.
Update	Fixed in commit *318ab44f88f9b4025be214c633fd3ca0a3d6ab0b*.

Title	Single access control role

ID	L-04
Description	The Permission Manager contract only has one owner role that has the ability to pause the contract, change owners, rotate the Coinbase cosigner key and other admin functionalities. Because a single owner key exists for all admin privileges, responses such as pausing the smart contract in the event of a key compromise will not exist.
Recommendation	We are aware of this issue and have chosen to keep one owner role, rather than using a separate key for each role's functionality.

Title	Signing key rate limiting for Coinbase cosigner
ID	**L-05**
Description	The Coinbase cosigner needs to have high availability and throughput at times where there is high demand for session key transactions. Because each user operation needs to be signed by the Coinbase cosigner or pending cosigner key, the key signing service should be able to handle a large volume of session key transactions.
Recommendation	Consider using a key signing service that is able to handle a high throughput of transactions to sign.

Title	Users can self-DOS by calling the useRecurringAllowance function
ID	**L-06**
Description	Users can self-DOS by calling the useRecurringAllowance function. Calling useRecurringAllowance with a callsSpend value greater than zero would deduct from the user's allowed spend for the current cycle, even if no funds have been spent. Additionally, the useRecurringAllowance function does not check if the permission is revoked when called directly by the user.
Recommendation	Consider running simulations to detect if user operations are having users call userRecurringAllowance in a non-session key transaction

Informational

ID	Description
I-01	**Gas Savings:**

	In approvePermission, we can return early if the permission is approved, rather than having to check the signature first. Consider switching the order of the checks to check if the permission is approved first before validating signatures. Fixed in commit *19dc3421ca46bb40a0a512c2f694111f0822d4e0*.
I-02	permissionManager and magicSpend values lack address(0) checks when set in the constructor in the PermissionCallableAllowedContractNativeTokenRecurringAllowance contract.
I-03	For incident responses, the permissionManager in the PermissionCallableAllowedContractNativeTokenRecurringAllowance contract is immutable. A compromised and redeployment in the Permission Manager contract would result in a redeployment of each individual permission contract.
I-04	ValueNotAllowed error is not used.
I-05	getRecurringAllowance and getRecurringAllowanceUsage do not take into consideration if the given permission has been revoked. As a result, an additional call to check if the permission has been revoked would need to be made to ensure that the returned allowance and usage is still usable.
I-06	A cycle usage's end time can overflow if the start + period is greater than the max value of type unit48. In a situation where an allowance's period is extremely high (near the unit48 max value), the permissioned allowance would not be usable. Fixed in PR #33.

Smart Wallet Reentrancy

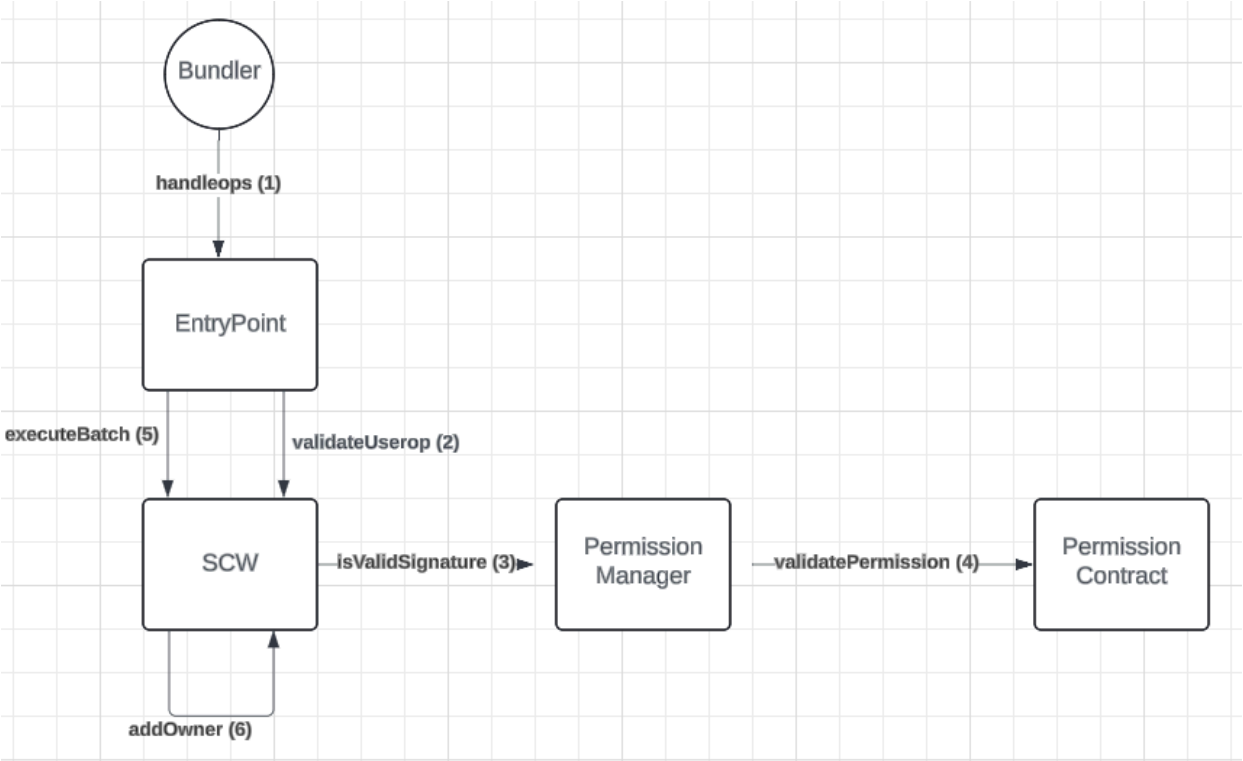
There are a few ways in which reentrancy back into the smart wallet account can be a problem. These scenarios include the following:

1. ***When the smart wallet is the caller and does a self call back into itself***. This would allow it to hit critical paths and call functions such as adding or removing owners on the smart wallet
2. ***When the smart wallet calls into an address that is an owner on the smart wallet, which the owner then calls back into the smart wallet***. This would also allow it to hit critical functionality because the caller who is reentering the smart wallet is an owner.
3. ***When the smart wallet calls into a contract who is not an owner, but this contract then calls into an owner of the smart wallet, which ultimately reenters the original smart wallet account***. This would also allow it to hit critical functionality because the caller who is reentering the smart wallet is an owner.

As a result, any call from an owner back into the smart wallet account is dangerous, regardless of any previous calls or the degree of separation between the previous calls. The risk of direct reentrancy is mitigated by not allowing the smart wallet account to call back into itself through

checks at the Permission Manager level. However, the risk of indirect reentrancy (#2 and #3) still exists.

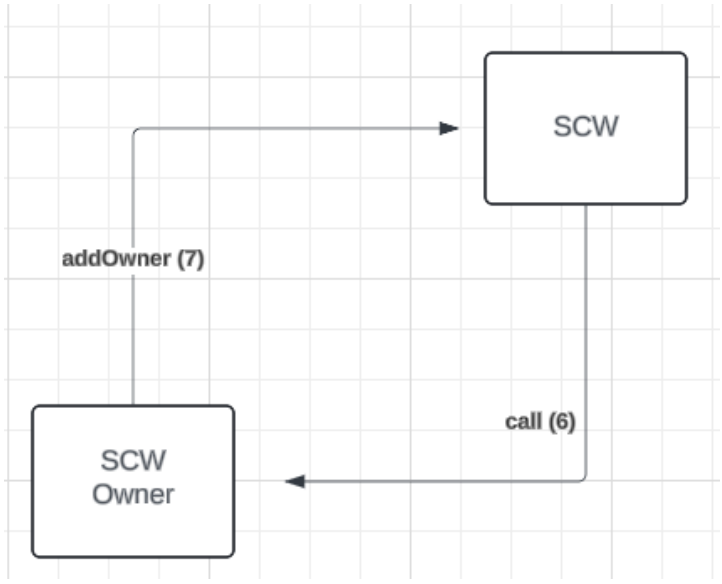
(1) SCW does a self call:



1. Bundler calls handleOps on entry point
2. Entrypoint calls validateUserop on the SCW
3. IsValidSignature on the Permission Manager is called (by the SCW) to validate the signature
4. Permission Manager then calls validatePermission on the Permission Contract to validate the individual permission at a more granular level. During this check, we ensure that for each of the Calls in the executeBatch function, only the permissionedCall function selector is able to be called (if the call target is not magic spend)
5. If all permission checks pass, we now enter the execution phase and executeBatch is executed with the given Calls
6. During the call, the SCW reenters itself by calling to add an arbitrary owner address.

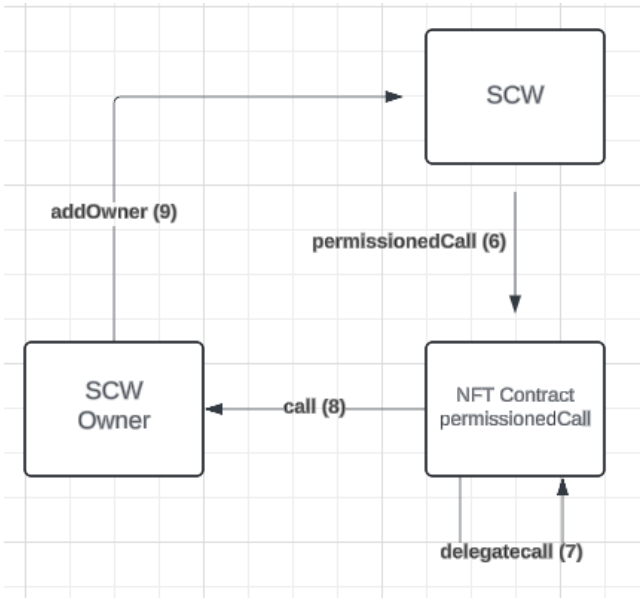
In scenario 1, we can observe that direct reentrancy can be dangerous because the onlyOwner check on the smart wallet can be bypassed if it is the smart wallet that is calling itself. Admin functionality can be invoked, such as adding or removing arbitrary owner addresses.

(2) SCW calls into an owner contract, which then reenters SCW



Even with one degree of separation, if the SCW calls into an address that is an owner on the smart wallet account, which then calls back into the SCW can be dangerous and invoke admin functionality.

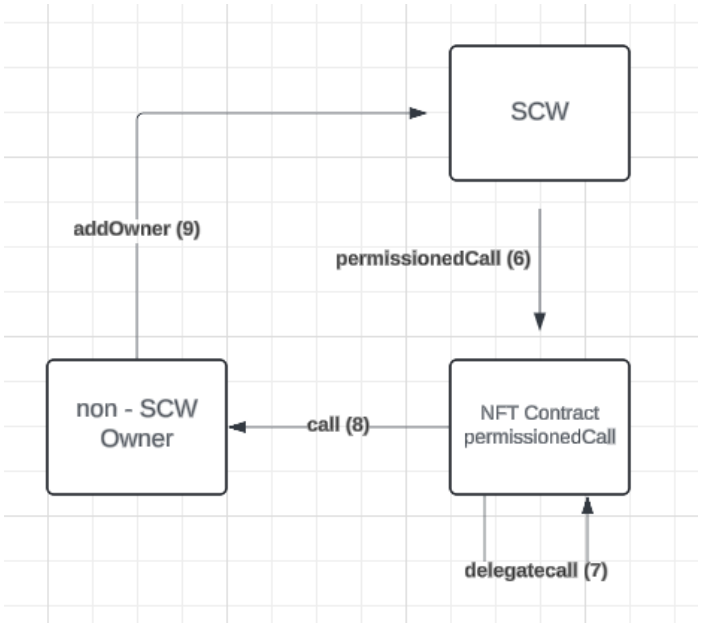
(3) SCW calls into non-owner contract, which then calls an owner that reenters SCW



Although permissioned contracts must implement the permissionCallable interface in order to be session key compatible, there is no check to enforce that the call in the allowed contract implementing the permissionCallable interface does a delegatecall (although the example contract recommends it). Regardless of whether we are doing a delegatecall or call on the NFT

contract, the same issue arises when the subsequent call calls into the SCW owner and then back into the smart wallet account.

(4) Non-SCW owner calls into SCW



In scenario 4, the difference here is that the address who calls back into the SCW in step 9 ***is not*** an owner of the SCW. Because of this, the call to addOwner would revert and not be valid.

Permission Manager Reentrancy

At the Permission Manager contract level, there also exists a reentrancy vector on both the approvePermission and revokePermission functions. Because permissions are able to be approved without a signature when the caller is the account the permission is approved for, calls that re-enter back into the Permission Manager contract are also dangerous. For example, if a Dapp constructs the following array of Calls that is passed into executeBatch, it could arbitrary approve and revoke permissions on behalf of of the smart wallet:

```
Unset
executeBatch(Calls[beforecalls, approvePermission,revokePermission,
permissionedCall, useRecurringAllowance])
```

This is gated at both the individual Permission contract level and on the Permission Manager contract. On the individual Permission contract, the call is gated by function selector and target addresses. Only allowed contract addresses and function selector specified by the permission

are allowed. Additionally, the Permission Manager also guards against self calls and does not allow calls beyond the first call (beforeCalls) to be on the Permission Manager.

User Operation Failures During Execution Phase

There are various situations where user operations may pass during the validation phase, but fail during execution (as highlighted in issues ***M-01*** and ***M-05***). For our first session key permission (Native Token Recurring Allowance), a paymaster is always required for each user operation. Because of this, any failed user operation and gas costs that will still need to be paid are offloaded to the Dapp's paymaster, rather than the user.

Trust Assumptions

There are several trust assumptions that are made in regards to the ***Coinbase cosigner***. These include the following:

1. ***The cosigner is able to detect outbound transfers of ERC20, ERC721 and ERC1155 tokens within user operations***. Although the Native Recurring Allowance permission contract does not explicitly allow ERC20 allowances as permissions, it is still possible for a permissioned contract to transfer ERC20 tokens on behalf of a user during the execution of its calls, given the user has previously approved the permissioned contract an allowance. It is a trust assumption that the Coinbase cosigner will simulate user operations that involve outbound token transfers of the aforementioned token standards and not sign them if so.
2. ***Setting honest gas values***. The gas values within a user operation that uses permissioned calls are set by Coinbase before it is submitted to a Bundler. It is assumed that these gas values are within a reasonable range.
3. ***Not signing user operations to sanctioned addresses if required***. If Coinbase is obligated through legal and compliance standards to not sign user operations that send funds to sanctioned addresses such as tornado cash, it is assumed that the cosigner will not sign such user operations.
4. ***The cosigner is available***. It is assumed that the Coinbase cosigner has high availability and will sign permitted user operations within a timely manner.