Coinbase: Spend Permissions

Security Review

Cantina Managed review by:

Riley Holterhus, Lead Security Researcher

Cccz, Security Researcher

December 10, 2024

Contents

1 Introduction

1.1 About Cantina

Cantina is a security services marketplace that connects top security researchers and solutions with clients. Learn more at cantina.xyz

1.2 Disclaimer

Cantina Managed provides a detailed evaluation of the security posture of the code at a particular moment based on the information available at the time of the review. While Cantina Managed endeavors to identify and disclose all potential security issues, it cannot guarantee that every vulnerability will be detected or that the code will be entirely secure against all possible attacks. The assessment is conducted based on the specific commit and version of the code provided. Any subsequent modifications to the code may introduce new vulnerabilities that were absent during the initial review. Therefore, any changes made to the code require a new security review to ensure that the code remains secure. Please be advised that the Cantina Managed security review is not a replacement for continuous security measures such as penetration testing, vulnerability scanning, and regular code reviews.

1.3 Risk assessment

Severity	Description
Critical	*Must* fix as soon as possible (if already deployed).
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.
Gas Optimization	Suggestions around gas saving practices.
Informational	Suggestions around best practices or readability.

1.3.1 Severity Classification

The severity of security issues found during the security review is categorized based on the above table. Critical findings have a high likelihood of being exploited and must be addressed immediately. High findings are almost certain to occur, easy to perform, or not easy but highly incentivized thus must be fixed as soon as possible.

Medium findings are conditionally possible or incentivized but are still relatively likely to occur and should be addressed. Low findings a rare combination of circumstances to exploit, or offer little to no incentive to exploit but are recommended to be addressed.

Lastly, some findings might represent objective improvements that should be addressed but do not impact the project's overall security (Gas and Informational findings).

2 Security Review Summary

Spend Permissions are a feature built on top of Coinbase's Smart Wallet contracts. They allow users to configure and manage permissions, enabling apps to spend native or ERC20 tokens on their behalf.

From Nov 27th to Dec 1st the Cantina team conducted a review of spend-permissions-review-1126 on commit hash c4053967. The team identified a total of **9** issues in the following risk categories:

- Critical Risk: 0
- High Risk: 0
- Medium Risk: 0
- Low Risk: 1
- Gas Optimizations: 0
- Informational: 8

3 Findings

3.1 Low Risk

3.1.1 `spendWithWithdraw()` may be using `WithdrawRequest` unexpectedly

Severity: Low Risk

Context: SpendPermissionManager.sol#L417-L443

Description: The `spendWithWithdraw()` function enables a spender to combine a spend operation with a `MagicSpend` withdrawal initiated from the spend permission's account. To validate that the `WithdrawRequest` corresponds to the `SpendPermission`, the function checks that both of their assets match and that the spend value equals the withdrawal amount:

```
function spendWithWithdraw(
    SpendPermission memory spendPermission,
    uint160 value,
    MagicSpend.WithdrawRequest memory withdrawRequest
) external requireSender(spendPermission.spender) {
    if (
        !(spendPermission.token == NATIVE_TOKEN && withdrawRequest.asset == address(0))
            && spendPermission.token != withdrawRequest.asset
    ) {
        revert SpendTokenWithdrawAssetMismatch(spendPermission.token, withdrawRequest.asset);
    }

    if (value != withdrawRequest.amount) {
        revert SpendValueWithdrawAmountMismatch(value, withdrawRequest.amount);
    }
    // ...
}
```

These checks may not fully prevent the `spendPermission.spender` from using a `WithdrawRequest` that was not intended for their spend permission.

For example, consider a scenario where the spend permission account is using `MagicSpend` for its ERC-4337 paymaster functionality. The account would receive an ETH `WithdrawRequest`, and this `WithdrawRequest` would not be expected to interact with the `SpendPermissionManager` at all.

However, a spender with ETH permissions could frontrun the ERC-4337 transaction and use the `WithdrawRequest` within the `SpendPermissionManager`, provided that they set the `value` to match the `withdrawRequest.amount`. This would consume the `WithdrawRequest`, which would break the ERC-4337 transaction and would require a new `WithdrawRequest` be generated.

Recommendation: Consider whether preventing the unexpected use of `WithdrawRequest` values within the `SpendPermissionManager` is important. If this is a concern, one potential mitigation is to require that a `WithdrawRequest` intended for the `SpendPermissionManager` encodes this intent within its `nonce` value.

Coinbase: Accepting the recommendation by encoding bits from the hash of the permission in the withdrawRequest nonce and validating within `spendWithWithdraw()`. Fixed initially in PR 49 with a follow-up change in PR 52.

Cantina Managed: Verified.

3.2 Informational

3.2.1 Signature replay considerations

Severity: Informational

Context: SpendPermissionManager.sol#L274-L309

Description: The `SpendPermissionManager` uses signatures in the `approveWithSignature()` and `approveBatchWithSignature()` functions. These functions do not include signature replay protection, but this is likely intentional. Replaying an approval multiple times using the same signature does not alter the state of the `SpendPermissionManager` or result in unintended consequences.

It's also worth noting that signature replay protection can safely be ignored because `_isRevoked` takes precedence over `_isApproved`. If this precedence were reversed (and approvals could override previous revokes) signature replay protection would become more important.

Recommendation: This finding has been provided for informational purposes and no code changes are necessary. Consider documenting this behavior in the code comments or documentation.

Coinbase: Acknowledged, won't fix as we don't track signatures and replays are idempotent.

Cantina Managed: Acknowledged.

3.2.2 `NATIVE_TOKEN` and `address(0)` edge cases could also trigger `SpendTokenWithdrawAssetMismatch`

Severity: Informational

Context: SpendPermissionManager.sol#L423-L428

Description: In the `spendWithWithdraw()` function, the following logic ensures that `withdrawRequest.asset` (the asset being withdrawn from `MagicSpend`) matches `spendPermission.token` (the asset being used in the `SpendPermissionManager`):

```
if (
    !(spendPermission.token == NATIVE_TOKEN && withdrawRequest.asset == address(0))
        && spendPermission.token != withdrawRequest.asset
) {
    revert SpendTokenWithdrawAssetMismatch(spendPermission.token, withdrawRequest.asset);
}
```

Note that this logic does not trigger the `SpendTokenWithdrawAssetMismatch` error in two edge cases:

1. `spendPermission.token == withdrawRequest.asset == NATIVE_TOKEN`.
2. `spendPermission.token == withdrawRequest.asset == address(0)`.

These two cases are potential candidates for throwing the error, as `MagicSpend` and `SpendPermissionManager` handle native token addresses differently, meaning a match between those values is unintended.

In the current behavior, both edge cases will eventually cause the code to revert. Specifically, `withdrawRequest.asset == NATIVE_TOKEN` is invalid in `MagicSpend` (and shouldn't have a corresponding signature) and `spendPermission.token == address(0)` is invalid in `SpendPermissionManager` (and is disallowed from ever being approved).

Recommendation: Consider whether these edge cases warrant updating the implementation to trigger the `SpendTokenWithdrawAssetMismatch` error directly. If not, consider documenting this behavior.

Coinbase: Acknowledged. Won't address due to lack of risk to user funds.

Cantina Managed: Acknowledged.

3.2.3 `spendWithWithdraw()` cannot spend smart wallet balances

Severity: Informational

Context: SpendPermissionManager.sol#L431

Description: `spendWithWithdraw()` will first withdraw funds from `MagicSpend` and then spend them. And it requires `withdrawRequest.amount` is equals to the `value` to spend:

```
if (value != withdrawRequest.amount) {
    revert SpendValueWithdrawAmountMismatch(value, withdrawRequest.amount);
}
```

Consider a user with 3 USDC in `MagicSpend`, 2 USDC in the smart wallet, and the app is ready to spend 5 USDC. Due to this check, `spendWithWithdraw()` can't use them to cover the 5 USDC spend.

One guess is that this check is to prevent spender from using `MagicSpend` unexpectedly, for example, `withdrawRequest1` is 3 USDC, `withdrawRequest2` is 5 USDC, App1's allowance is 3 USDC, and App2's allowance is 5 USDC. So App1 will not be able to use `withdrawRequest2`, and if App2 first consumes 3 USDC with `withdrawRequest1`, it will not be able to use `withdrawRequest2` ($3 + 5 > 5$).

If so, the `if (value < withdrawRequest.amount)` check will also work for the above case, and allow App to use `MagicSpend` + smart wallet balance.

Recommendation: It is recommended to change to

```
- if (value != withdrawRequest.amount) {
+ if (value < withdrawRequest.amount) {
      revert SpendValueWithdrawAmountMismatch(value, withdrawRequest.amount);
  }
```

Coinbase: Accepting recommendation. Fixed in PR 50.

Cantina Managed: Verified.

3.2.4 `approveBatchWithSignature()` may revert due to malformed `spendPermission`

Severity: Informational

Context: SpendPermissionManager.sol#L314-L333

Description: `approveBatchWithSignature()` approves a batch of spendPermissions. And when one of those spendPermissions has been revoked, the function doesn't revert, it just sets `allApproved` to `false`.

But if App can add malformed spendPermission to make `approveBatchWithSignature()` fail so that even the normal spendPermission can't be approved.

```
function _approve(SpendPermission memory spendPermission) internal returns (bool) {
    // check token is non-zero
    if (spendPermission.token == address(0)) revert ZeroToken();

    // check spender is non-zero
    if (spendPermission.spender == address(0)) revert ZeroSpender();

    // check period non-zero
    if (spendPermission.period == 0) revert ZeroPeriod();

    // check allowance non-zero
    if (spendPermission.allowance == 0) revert ZeroAllowance();

    // check start is strictly before end
    if (spendPermission.start >= spendPermission.end) {
        revert InvalidStartEnd(spendPermission.start, spendPermission.end);
    }
```

For example, App1, App2 provide correctly formatted spendPermission1 and spendPermission2, but App3 provides wrongly formatted spendPermission3 where allowance == 0. When they are approved by `approveBatchWithSignature()`, `_approve(spendPermission3)` will revert so that spendPermission1 and spendPermission2 cannot be approved either.

Recommendation: Considering check the spendPermission format in `getBatchHash()` so that incorrect spendPermissions can be weeded out when the user fetches the data to sign. And also can juct check it on the front-end.

Coinbase: Acknowledged, won't fix. (we don't expect cross-app permissions to appear in a shared batch).

Cantina Managed: Acknowledged.

3.2.5 Permissions can be approved for ERC-721 tokens with unintended effects

Severity: Informational

Context: SpendPermissionManager.sol#L717

Description: When a spend permission is executed, the `_transferFrom()` function ultimately calls the `transferFrom(address,address,uint256)` function on the `token` address (via the `SafeTransferLib` library). It is intended for the `token` address to be an ERC20 contract, however ERC721 contracts also implement a function with the same `transferFrom(address,address,uint256)` signature. In the case of ERC721 contracts, the `uint256` argument represents a token ID instead of an amount.

Since the `SpendPermissionManager` is not designed to support ERC721 tokens, it may be preferable to explicitly prevent permissions from being created with an ERC721 token.

Note: This issue was raised by the Coinbase team and is included here for tracking purposes.

Recommendation: Consider explicitly preventing ERC721 tokens from being used in the `SpendPermissionManager`. One approach would be using ERC165 to check if a contract implements the ERC721 interface and revert if so. However, note that not all ERC721 contracts support ERC165, so this solution may not cover every scenario.

Coinbase: Fixed in PR 51 and PR 55, with a refactor to use OpenZeppelin's `ERC165Checker` in PR 56.

Cantina Managed: Verified.

3.2.6 `isValidERC6492SignatureNowAllowSideEffects()` signature length padding behavior

Severity: Informational

Context: PublicERC6492Validator.sol#L26

Description: In the `isValidERC6492SignatureNowAllowSideEffects()` function, the following code is checking whether the last 32 bytes of the signature match the magic bytes `0x6492649264926492649264926492649264926492649264926492649264926492`:

```
function isValidERC6492SignatureNowAllowSideEffects(
    address signer,
    bytes32 hash,
    bytes memory signature
) internal returns (bool isValid) {
    /// @solidity memory-safe-assembly
    assembly {
        // ...
        for { let n := mload(signature) } 1 {} {
            if iszero(eq(mload(add(signature, n)), mul(0x6492, div(not(isValid), 0xffff)))) {
                isValid := callIsValidSignature(signer, hash, signature)
                break
            }
            // ...
        }
    }
}
```

Note that this implementation does not verify that `signature.length >= 32` before this check, and as a result, the memory loaded in with `mload(add(signature, n))` might not be from the signature content, and instead the `mload()` could load memory relating to the signature's length.

To see this behavior, consider a scenario where `signature = hex"aa"` (which has length 1). One example of how the memory layout during `isValidERC6492SignatureNowAllowSideEffects()` could look is the following:

Location	Description of data	Data
0x00	scratch space	00 00 ... 00 00
0x20	scratch space	00 00 ... 00 00
0x40	free memory pointer	00 00 ... 00 c0
0x60	zero slot	00 00 ... 00 00
0x80	signature length	00 00 ... 00 01
0xa0	signature content	aa 00 ... 00 00

In this scenario, `add(signature, n)` would be `add(0x80, 1) == 0x81`, which means the `mload()` would load `00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 01 aa`. Notice that the first 31 bytes of this value are coming the signature's length, and aren't actually from the content of the signature.

Essentially, this behavior pads signatures shorter than 32 bytes with the least significant `32 - signature.length` bytes of the signature's length during the magic byte comparison.

Fortunately, this does not appear to lead to any issues. If `signature.length < 32`, then it should not be a match against the magic bytes, and this seems to always be the case. There can't be a false match, because `signature.length < 32` implies that the least-significant byte of the signature length is one of `0x00, 0x01, ..., 0x1f`, none of which are a match for `0x64` or `0x92` from the magic bytes.

Note: this finding relates to the `isValidERC6492SignatureNowAllowSideEffects()` function within the Solady `SignatureCheckerLib`, which is a dependency of the codebase.

Recommendation: This finding has been provided for informational purposes and no code changes are necessary. If changes to the library function are possible, consider documenting this behavior in the code comments or documentation.

Coinbase: Acknowledged, will share with Solady author.

Cantina Managed: Acknowledged.

3.2.7 `isValidERC6492SignatureNowAllowSideEffects()` can differ from reference ERC-6492 implementation

Severity: Informational

Context: PublicERC6492Validator.sol#L26

Description: When deploying a contract using ERC-6492, the following scenario can occur:

- The signer address does not initially have code.
- The deployment call succeeds in deploying the signer.
- The first call to `isValidSignature()` fails.

In this case, notice that `isValidERC6492SignatureNowAllowSideEffects()` continues by making a second external call and then another `isValidSignature()` call:

```
function isValidERC6492SignatureNowAllowSideEffects(
    address signer,
    bytes32 hash,
    bytes memory signature
) internal returns (bool isValid) {
    /// @solidity memory-safe-assembly
    assembly {
        // ...
        for /* ... */ {
            // ...
            if iszero(extcodesize(signer)) {
                if iszero(call(gas(), mload(o), 0, add(d, 0x20), mload(d), codesize(), 0x00)) {
                    break
                }
            }
            // ...
            isValid := callIsValidSignature(signer, hash, s)
            if iszero(isValid) {
                if call(gas(), mload(o), 0, add(d, 0x20), mload(d), codesize(), 0x00) {
                    isValid := callIsValidSignature(signer, hash, s)
                }
            }
            break
        }
    }
}
```

This behavior differs from the reference implementation of ERC-6492. This can be seen since recursive calls to `isValidSigImpl()` are only made if `contractCodeLen > 0` (i.e. if the contract was not deployed during the ERC-6492 logic):

```
function isValidSigImpl(
    address _signer,
    bytes32 _hash,
    bytes calldata _signature,
    bool allowSideEffects,
    bool tryPrepare
) public returns (bool) {
    uint contractCodeLen = address(_signer).code.length;
    // ...
    if (isCounterfactual) {
      // ...
      if (contractCodeLen == 0 || tryPrepare) {
        (bool success, bytes memory err) = create2Factory.call(factoryCalldata);
        if (!success) revert ERC6492DeployFailed(err);
      }
    } else {
      sigToValidate = _signature;
    }
    // ...
    if (isCounterfactual || contractCodeLen > 0) {
      try IERC1271Wallet(_signer).isValidSignature(_hash, sigToValidate) returns (bytes4 magicValue) {
        // ...
        if (!isValid && !tryPrepare && contractCodeLen > 0) {
          return isValidSigImpl(_signer, _hash, _signature, allowSideEffects, true);
        }
        // ...
      } catch (bytes memory err) {
        // ...
        if (!tryPrepare && contractCodeLen > 0) {
          return isValidSigImpl(_signer, _hash, _signature, allowSideEffects, true);
        }
        // ...
      }
    }
    // ...
}
```

This discrepancy does not seem to cause any major issues. It is unlikely that the current behavior of `isValidERC6492SignatureNowAllowSideEffects()` is useful, as the second external call and the second `isValidSignature()` call will likely fail if the first `isValidSignature()` call fails after deployment.

Note: this finding relates to the `isValidERC6492SignatureNowAllowSideEffects()` function within the Solady

`SignatureCheckerLib`, which is a dependency of the codebase.

Recommendation: If changes to the downstream `Solady` code are possible, consider aligning with the reference implementation behavior to avoid redundant calls in this scenario. One possible implementation of this is the following:

```
  function isValidERC6492SignatureNowAllowSideEffects(
      address signer,
      bytes32 hash,
      bytes memory signature
  ) internal returns (bool isValid) {
      /// @solidity memory-safe-assembly
      assembly {
          // ...
          for /* ... */ {
              // ...
-             if iszero(extcodesize(signer)) {
+             let initialCodesizeIsZero := iszero(extcodesize(signer))
+             if initialCodesizeIsZero { // if initial codesize is zero
                  if iszero(call(gas(), mload(o), 0, add(d, 0x20), mload(d), codesize(), 0x00)) {
                      break
                  }
              }
              // ...
              isValid := callIsValidSignature(signer, hash, s)
-             if iszero(isValid) {
+             if and(iszero(isValid), iszero(initialCodesizeIsZero)) { // if first call not valid and initial
↪   codesize was not zero
                  if call(gas(), mload(o), 0, add(d, 0x20), mload(d), codesize(), 0x00) {
                      isValid := callIsValidSignature(signer, hash, s)
                  }
              }
              break
          }
      }
  }
```

Coinbase: Acknowledged, will share with author of Solady.

Cantina Managed: Acknowledged.

3.2.8 ERC-6492 `factory`/`prepareTo` address considerations

Severity: Informational

Context: PublicERC6492Validator.sol#L26

Description: The `isValidERC6492SignatureNowAllowSideEffects()` function decodes the `factory`/`prepareTo` address using the following code:

```
function isValidERC6492SignatureNowAllowSideEffects(
    address signer,
    bytes32 hash,
    bytes memory signature
) internal returns (bool isValid) {
    /// @solidity memory-safe-assembly
    assembly {
        // ...
        for /* ... */ {
            // ...
            let o := add(signature, 0x20) // Signature bytes.
            // ...
            if /* ... */ {
                if iszero(call(/* ... */, mload(o), /* ... */) {
                    break
                }
            }
            // ...
            if /* ... */ {
                if call(/* ... */, mload(o), /* ... */) {
                    // ...
                }
            }
            // ...
        }
    }
}
```

Two insights are relevant to the above code:

1. The comment "`Signature bytes`" is somewhat vague, and could be improved by replacing the comment with "`factory/prepareTo address`" instead.
2. Since the code uses assembly to decode addresses, it does not revert or change its behavior if the addresses have dirty upper bits. For example, notice that the following modification to the `test/base/SpendPermissionManagerBase.sol` test function does not affect any test behavior:

```
    function _signSpendPermission6492(
        SpendPermissionManager.SpendPermission memory spendPermission,
        uint256 ownerPk,
        uint256 ownerIndex,
        bytes[] memory allInitialOwners
    ) internal view returns (bytes memory) {
        // ...
-       bytes memory eip6492Signature = abi.encode(factory, factoryCallData, wrappedSignature);
+       bytes memory eip6492Signature = abi.encode(
+           uint256(uint256(uint160(factory)) |
↪   0x1111111111111111111111110000000000000000000000000000000000000000),
+           factoryCallData,
+           wrappedSignature
+       );
        // ...
    }
```

   Fortunately, this does not appear to be a concern, especially because there are many other ways for someone to slightly modify a signature while still maintaining the signature's validity. This is why it's a common best practice recommendation to avoid using signatures as unique identifiers. There is more material about this topic in Kadenzipfel's article about signature malleability.

Note: this finding relates to the `isValidERC6492SignatureNowAllowSideEffects()` function within the Solady `SignatureCheckerLib`, which is a dependency of the codebase.

Recommendation: If downstream changes to the `Solady` library are possible, consider updating the comment referenced above, and also consider documenting the behavior of dirty bits in the `factory/prepareTo` address.

Coinbase: Acknowledged. Won't address, but will share all Solday-related findings with the author of the library.

Cantina Managed: Acknowledged.