CANTINA

Coinbase: Spend Permissions

Competition

January 5, 2025

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

A competition provides a broad evaluation of the security posture of the code at a particular moment based on the information available at the time of the review. While competitions endeavor to identify and disclose all potential security issues, they 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, therefore, any changes made to the code would require an additional security review. Please be advised that competitions are 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 30th to Nov 6th Cantina hosted a competition based on spend-permissions. The participants identified a total of **17** issues in the following risk categories:

- Critical Risk: 0
- High Risk: 0
- Medium Risk: 0
- Low Risk: 6
- Gas Optimizations: 0
- Informational: 11

3 Findings

3.1 Low Risk

3.1.1 The SpendPermissionManger Fails To Handle The Case When It Is Removed From Wallet Owner List, Causing DoS For Dapps

Submitted by JesJupyter

Severity: Low Risk

Context: SpendPermissionManager.sol#L365-L368

Description: The isApproved function checks if a spend permission for an account is still valid by determining whether it has been approved and not revoked.

It is primarily used within _useSpendPermission to validate the spend permissions before executing any transaction.

Additionally, external DApps may rely on isApproved to ensure that a permission remains active, thereby preventing unnecessary Denial-of-Service (DoS) attempts or failed transactions due to expired or revoked permissions.

```
function isApproved(SpendPermission memory spendPermission) public view returns (bool) {
    bytes32 hash = getHash(spendPermission);
    return !_isRevoked[hash][spendPermission.account] && _isApproved[hash][spendPermission.account];
}
```

The above implementation only checks whether the spend permission was previously approved and not explicitly revoked. However, this logic does not account for cases where users may decide to revoke access to the SpendPermissionManager indirectly, by simply removing the contract's address from the list of owner addresses associated with their Coinbase Smart Wallet.

If the user does this without calling revoke directly, the spend permission appears to remain valid. Consequently, isApproved would still return true, potentially leading to erroneous assumptions by DApps, which may attempt to perform transactions under the mistaken belief that permission is still intact. This can result in transaction failures, DoS, and gas costs wasted on failed transactions.

Recommendation: To address this vulnerability, it is advisable to enhance isApproved by adding an additional check to ensure that SpendPermissionManager is still recognized as an owner in the user's wallet. This adjustment would allow isApproved to return false when the manager has been removed from the owner's list, thereby preventing the DApp from mistakenly assuming valid permissions.

```
function isApproved(SpendPermission memory spendPermission) public view returns (bool) {
    bytes32 hash = getHash(spendPermission);
    return CoinbaseSmartWallet(spendPermission.account).isOwnerAddress(address(this)) &&
↪   !_isRevoked[hash][spendPermission.account] && _isApproved[hash][spendPermission.account];
}
```

3.1.2 Prevent the approval or revoke of spend permissions that are in an incoherent status

Submitted by StErMi, also found by catchme, newspacexyz, merlin, yashar, KevinKKien, Bigsam, OrangeSantra, Xavek, Sujith Somraaj, Sujith Somraaj, armormadeofwoe, radeveth and dobrevaleri

Severity: Low Risk

Context: SpendPermissionManager.sol#L289-L290, SpendPermissionManager.sol#L439-L440

Description: The current implementation of both _approve and revoke should be further improved by reverting when the current state of the spend permission is incompatible with the request.

- _approve should revert if the spend permission has been already approved.
- _approve should revert if the spend permission has been already revoked.
- revoke should revert if the spend permission has been already revoked.
- revoke should revert if the spend permission has not yet been approved.

The above suggestions will make the state of the contract much more coherent and will prevent the emission of inconsistent events.

Recommendation: Coinbase should implement the above sanity checks inside the `_approve` and `revoke` functions.

3.1.3 A malicious account can steal from app or spender by using a smart contract that has empty execute function

Submitted by Joshuajee, also found by JesJupyter, gesha17 and karanel

Severity: Low Risk

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

Description: The spend permission is supposed to work with Coinbase smart accounts, but anyone can create a malicious Smart Wallet with the same interface as Coinbase Smart Wallet, this wallet will be used to trick apps and use their services for free.

Apps use the spend permission by calling the `spend`, this makes an internal call to the `_transferFrom` function, which calls the internal `execute` function.

```
function _execute(address account, address target, uint256 value, bytes memory data) internal virtual {
    CoinbaseSmartWallet(payable(account)).execute({target: target, value: value, data: data});
}
```

This calls the `execute` function on the Smart Wallet, on a real Smart Wallet the execute function makes the necessary calls to transfer the funds to the recipient, but for the Malicious Smart Wallet, the execute function will do nothing.

So when the app calls spend, the transaction will go through without revert and the app will give value to the malicious user thinking it has received the funds.

Impact: This will lead to a loss of funds to Apps or Spenders that blindly trust the spend, so this is a High Impact.

Recommendation: Check the balance of the spender before and after calling execute to ensure that they received the funds.

```
  function spend(SpendPermission memory spendPermission, uint160 value)
      external
      requireSender(spendPermission.spender)
  {
      _useSpendPermission(spendPermission, value);
+    uint balanceBefore = spendPermission.token == NATIVE_TOKEN ? address(spendPermission.token).balance :
↪   spendPermission.token.balanceOf(spendPermission.account);
      _transferFrom(spendPermission.token, spendPermission.account, spendPermission.spender, value);
+    uint balanceAfter = spendPermission.token == NATIVE_TOKEN ? address(spendPermission.token).balance :
↪   spendPermission.token.balanceOf(spendPermission.account);
+    assert(balanceAfter <= balanceBefore + value);

  }
```

The downside of the above is that it may not work with Fee on Transfer tokens, so another fix is to ensure that the Apps do the check themselves.

3.1.4 Owner of a spendpermission can sign spenpermissions that will never be executed

Submitted by 0xHelium, also found by catchme, 0xBeastBoy, yashar, kind0dev, IvanFitro, shred, shred, 0xu-markhatab, mahivasisth, 0xpetern, Mosh, karanel, S0x0mtee, almantare and *arman*

Severity: Low Risk

Context: SpendPermissionManager.sol#L34

Description: Suppose Bob has to pay Alice for a service, so Bob issues a spendpermission to Alice so that she will get paid. The problem is that there is no sufficient mechanism to ensure Bob's spendpermission can be spent, and therefore Bob can issue invalid spendpermissions and approve or sign them, but Alice will never be able to spend the approved tokens. Consider this scenario:

- Current time is > `spendpermission.end`.
- Bob sign a permission that starts and ends in the past.
- When Alice attempts to spend the tokens, the transaction will revert.

Proof of Concept: Copy and paste this test into approveWithSignature.t.sol

```
function test_approveBadPermission_DoS_Spender(
    address spender,
    //address token,
    uint48 start,
    uint48 end,
    uint48 period,
    uint160 allowance,
    uint256 salt,
    bytes memory extraData
) public {
    vm.assume(spender != address(0));
    //vm.assume(token != address(0));
    vm.assume(start < end);
    vm.assume(period > 0);
    vm.assume(allowance > 0);

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

    vm.deal(address(account),allowance);
    assertEq(address(account).balance, allowance);

    vm.startPrank(address(account));
    mockSpendPermissionManager.approve(spendPermission);
    vm.assertTrue(mockSpendPermissionManager.isApproved(spendPermission));
    vm.stopPrank();

    // spend the permission
    vm.warp(start);

    vm.startPrank(address(spender));
    mockSpendPermissionManager.spend(spendPermission, allowance);
    vm.stopPrank();

    vm.warp(end);

    //sign the same permission so that spending will be DosEd
    bytes memory signature = _signSpendPermission(spendPermission, ownerPk, 0);
    mockSpendPermissionManager.approveWithSignature(spendPermission, signature);
    vm.assertTrue(mockSpendPermissionManager.isApproved(spendPermission));

    vm.deal(address(account),allowance);
    assertEq(address(account).balance, allowance);

    vm.expectRevert();
    vm.startPrank(address(spender));
    mockSpendPermissionManager.spend(spendPermission, allowance);
}
```

Run it using `forge test --mt test_approveBadPermission_DoS_Spender`.

Recommendation: Before approving a spendpermission, make sure start and end are in the future.

3.1.5 `Spendpermission` can be create by account that have 0 tokens to spend

Submitted by 0xHelium, also found by 0xpetern, JesJupyter and cryptostaker

Severity: Low Risk

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

Description: Owner of a spend permission can create spendpermissions even if he have 0 token balance, effectively preventing spender from spending the permission. He can simply have 0 token or transfer tokens out of his wallet after approving.

Proof of Concept: Copy and paste this test in `approveWithSignature.t.sol`:

```
function test_approveBadPermission_0Tokenbalance(
    address spender,
    //address token,
    uint48 start,
    uint48 end,
    uint48 period,
    uint160 allowance,
    uint256 salt,
    bytes memory extraData
) public {
    vm.assume(spender != address(0));
    //vm.assume(token != address(0));
    vm.assume(start < end);
    vm.assume(period > 0);
    vm.assume(allowance > 0);

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

    assertEq(address(account).balance, 0);

    vm.startPrank(address(account));
    mockSpendPermissionManager.approve(spendPermission);
    vm.assertTrue(mockSpendPermissionManager.isApproved(spendPermission));
    vm.stopPrank();

    // spend the permission
    vm.warp(start);

    vm.startPrank(address(spender));
    vm.expectRevert();
    mockSpendPermissionManager.spend(spendPermission, allowance);
    vm.stopPrank();
}
```

Run it with:

```
forge test --mt test_approveBadPermission_0Tokenbalance
```

Recommendation: Check owner balance to be >= allowance before approving a spend permission. This check is not sufficient because owner can approve with enough balance and then after withdraw the balance to another address to reproduce the attack. So a better approach is to transfer token from owner to the contract and then transfer it to spender afterwards.

3.1.6 `_transferFrom()` does not revert for not-yet-deployed tokens

Submitted by MiloTruck, also found by Blockdev, Bluedragon and mgf15

Severity: Low Risk

Context: SpendPermissionManager.sol#L492-L497, SpendPermissionManager.sol#L507-L509, CoinbaseSmartWallet.sol#L282-L289

Description: When performing token transfers in `_transferFrom()`, calldata for `IERC20.transfer` is forwarded to `CoinbaseSmartWallet.execute()`:

```
_execute({
    account: account,
    target: token,
    value: 0,
    data: abi.encodeWithSelector(IERC20.transfer.selector, recipient, value)
});
```

```
function _execute(address account, address target, uint256 value, bytes memory data) internal virtual {
    CoinbaseSmartWallet(payable(account)).execute({target: target, value: value, data: data});
}
```

`CoinbaseSmartWallet.execute()` executes the forwarded calldata with a low-level call and checks that it did not revert (ie. `success = true`):

```
function _call(address target, uint256 value, bytes memory data) internal {
    (bool success, bytes memory result) = target.call{value: value}(data);
    if (!success) {
        assembly ("memory-safe") {
            revert(add(result, 32), mload(result))
        }
    }
}
```

However, when a low-level `.call()` is performed with `target` as an address without code, `success` will also be `true`.

As a result, if spend permissions are added for a token that hasn't been deployed but its address is known ahead of time, `spend()` can be called to spend the allowance although the token isn't actually transferred. For example:

- A token will be deployed from a factory that uses `CREATE2`.
- Since the token address is deterministic and known ahead of time, a user precomputes the token's address and grants spend permissions for the token.
- If the spender calls `spend()` for that token before it is deployed, `spend()` will not revert.
- However, the token is not transferred as it hasn't been deployed.

An example of such a token would be Uniswap V2 LP tokens, which are deployed with `CREATE2` in `UniswapV2Factory`.

Recommendation: In `_transferFrom()`, check that the `token` address has code:

```
  } else {
+     if (token.code.length == 0) revert TokenHasNoCode();
      _execute({
          account: account,
          target: token,
          value: 0,
          data: abi.encodeWithSelector(IERC20.transfer.selector, recipient, value)
      });
  }
```

3.2 Informational

3.2.1 `Approve` function is no check the _isRevoked or not, make lead to `isApproved` function failed

Submitted by catchme, also found by 4gontuk, 0xtincion, cryptozaki, radeveth, 0xleadwizard and prapandey031

Severity: Informational

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

Description: The `Approve` function:

```
function _approve(SpendPermission memory spendPermission) internal {
    // 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);
    }

    bytes32 hash = getHash(spendPermission);
    _isApproved[hash][spendPermission.account] = true;
    emit SpendPermissionApproved(hash, spendPermission);
}
```

If we do not check whether the spend permission has been revoked when calling the `_approve` function, and we don't reset the revoked status, which will lead to the follow problem:

- Ineffective Approval: The spend permission will remain in a revoked state even after being approved again. This means any attempts to use this spend permission will fail because the system still considers it revoked.
- Irreversible Revocation: Without the ability to reset the revoked status, once a permission is revoked, it cannot be re-approved.
- Inconsistent State: The `_isApproved` mapping will indicate that the permission is approved, while the `_isRevoked` mapping indicates it is revoked. This inconsistency can lead to unexpected behavior in the contract logic.

We can see the isApproved function check `_isRevoked[hash][spendPermission.account]`, if in the `_approve`, we don't set `_isRevoked[hash][spendPermission.account]` to `false` (like the code below), it will lead to `isApproved` returning `false`, this is not a match our whole code logic.

```
function isApproved(SpendPermission memory spendPermission) public view returns (bool) {
    bytes32 hash = getHash(spendPermission);
    return !_isRevoked[hash][spendPermission.account] && _isApproved[hash][spendPermission.account];
}
```

Recommendation: We should set the `_isRevoked[hash][spendPermission.account]` to `false`, in the `_approve` function.

3.2.2 `SignatureCheckerLib` does not implement ERC-6492 verification

Submitted by frangio, also found by amaron and stanchev

Severity: Informational

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

ERC-6492 mandates the following verification procedure:

- check if the signature ends with magic bytes, in which case do an `eth_call` to a multicall contract that will call the factory first with the `factoryCalldata` and deploy the contract if it isn't already deployed; Then, call `contract.isValidSignature` as usual with the unwrapped signature
- check if there's contract code at the address. If so perform ERC-1271 verification as usual by invoking `isValidSignature`
- if the ERC-1271 verification fails, and the deploy call to the `factory` was skipped due to the wallet already having code, execute the `factoryCalldata` transaction and try `isValidSignature` again
- if there is no contract code at the address, try `ecrecover` verification

The mention of `eth_call` is only relevant for off-chain verification, but this is otherwise the procedure for on-chain verification as well, as corroborated by the Reference Implementation.

Solady's `SignatureCheckerLib` does not implement the last step, i.e., fallback to `ecrecover`.

Impact: In addition to being non-compliant, this error results in the practical consequence that `PublicERC6492Validator` is not usable with signatures by EOAs. Due to the lack of `ecrecover` fallback, `PublicERC6492Validator` will not be able to validate a signature unless it is a smart contract signature.

Recommendation: Add the missing `ecrecover` fallback in Solady's implementation of ERC-6492.

Alternatively, considering the risks associated to an assembly implementation, use a Solidity library such as the one created by the authors of ERC-6492, AmbireTech/signature-validator.

3.2.3 Batch Approval Duplicate Permissions Vulnerability in `approveBatchWithSignature`

Submitted by igdbase, also found by catchme, newspacexyz, kind0dev, ACai, 0xumarkhatab, trachev, bbash, Daniel526, Gaurav, newspacexyz and amaron

Severity: Informational

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

Impact: The `approveBatchWithSignature` function allows duplicate permissions within the same batch to be approved without any safeguards to detect and prevent them. This results in inefficiencies in processing (e.g., redundant gas costs) and increases the potential for unexpected behavior due to duplicate permission entries. Although accidental duplication is unlikely due to the unique `salt` and time-based parameters, intentional or user-error duplicates could still arise, particularly in systems where permissions are managed off-chain or user-defined salts are used. This oversight may lead to:

- Higher gas costs per transaction due to redundant processing.
- Increased transaction size, which may raise costs further or potentially lead to out-of-gas errors.
- Unexpected behavior or permission conflicts that could open paths for misuse or errors in multi-transaction workflows.

Proof of Concept: The vulnerability exists in the following function:

```
function approveBatchWithSignature(SpendPermission[] calldata permissions, bytes calldata signature) external {
    for (uint i = 0; i < permissions.length; i++) {
        SpendPermission memory permission = permissions[i];
        // Existing code processes each permission, without duplicate checks.
    }
}
```

In this function:

1. Lack of Duplicate Checking: There is no mechanism to prevent duplicate `SpendPermission` entries within a single batch.
2. Code Link: SpendPermissionManager.sol#L237

Consider a scenario where a batch includes two identical permissions, either accidentally or intentionally. Without a duplicate check, both entries are processed separately, incurring additional gas fees and increasing transaction size. If permissions involve user-defined salts, it could also lead to repeated approvals that may cause conflicting or unexpected results, especially if permissions have overlapping or dependent actions.

Recommendation: To address this vulnerability, implement a check for duplicates in the `approveBatchWithSignature` function:

1. Tracking Processed Permissions: Use a `mapping` or `set` to store each processed `permission` identifier within the batch.
2. Filter Out Duplicates: Before processing each `permission`, check if it exists in the `mapping`. If it does, skip the processing to prevent redundancy.
3. Example Implementation:

```
mapping(bytes32 => bool) private processedPermissions;

function approveBatchWithSignature(SpendPermission[] calldata permissions, bytes calldata signature) external {
    for (uint i = 0; i < permissions.length; i++) {
        bytes32 permissionHash = keccak256(abi.encode(permissions[i]));

        // Skip duplicate permissions
        if (processedPermissions[permissionHash]) continue;
        processedPermissions[permissionHash] = true;

        // Process permission as intended
    }
}
```

3.2.4 Should have check to ensure end and `starttime > currenttime`

Submitted by shred

Severity: Informational

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

Description: When approving this, there is no check to ensure end and `starttime > currenttime`, but in `getCurrentPeriod`, which is used in `spend`, there is a related check. This makes it possible to create useless spend approvals:

```
function getCurrentPeriod(SpendPermission memory spendPermission) public view returns (PeriodSpend memory) {
    // check current timestamp is within spend permission time range
    uint48 currentTimestamp = uint48(block.timestamp);
    if (currentTimestamp < spendPermission.start) { // <<<
        revert BeforeSpendPermissionStart(currentTimestamp, spendPermission.start);
    } else if (currentTimestamp >= spendPermission.end) {
        revert AfterSpendPermissionEnd(currentTimestamp, spendPermission.end);
    }
```

Recommendation: Add said check.

3.2.5 Inability to Pause or Update Version: Risk of Delayed Response to Critical Bugs

Submitted by Anurag Jain, also found by phil, catellatech and jesjupyter

Severity: Informational

Context: SpendPermissionManager.sol

Description: If any serious bug is discovered in the `SpendPermissionManager` contract, there is:

1. No immediate way for contract to pause all activities.
2. No way to revoke all offline signatures by upgrading version instantly.

This can put User funds to risk.

Impact: If a security bug is discovered in the future, upgrading the contract (in the case of a proxy) would be necessary. However, this process can be time-consuming, and no immediate action could be taken to mitigate the issue. As a result, there would be an increased risk of further fund losses.

Likelihood: Medium.

Recommendation:

1. Implement the `Pausable` contract from OpenZeppelin and restrict specific functions to only operate when the contract is unpaused.
2. Enable the owner to update the contract version via a setVersion function, and override EIP712.sol#L95 to always return true. This approach allows the owner to instantly invalidate all previously signed approvals if a bug is identified in the contract's allowance logic.

3.2.6 Lack of getter function for permissions

Submitted by phil

Severity: Informational

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

Description: The contract lacks a getter function to retrieve an account's permissions, or a specific permission details. This creates a usability barrier, especially for less sophisticated/technical users. Currently, users must have specific `SpendPermission` details to perform essential actions, such as calling `SpendPermissionManager::revoke()` to revoke a permission or `SpendPermissionManager::getCurrentPeriod()` to check the consumption of approved values within a period. Adding a getter function would improve ease of use, especially for users with limited technical knowledge.

The `spender-permissions` utility is designed to interact with smart wallets and will likely be used by a broad range of users, including those without extensive blockchain or programming experience. However, as of current implementation, users must manually input specific `SpendPermission` details when attempting to retrieve status information or revoke permissions. For instance:

- Checking permission usage: to check how much of the approved amount has been utilized within a given period, users need to call `SpendPermissionManager::getCurrentPeriod()` with the full struct details.
- Revoking permissions: users must call `SpendPermissionManager::revoke()` with the full struct details.

The absence of a straightforward getter function to list active permissions makes managing permissions much more complex. This might limit how useful or safe this utility might be for users.

Recommendation: Implement getter function(s) for permissions.

3.2.7 The PublicERC6492Validator won't work on all evm compatible chains due to reverting verifier not being deployed

Submitted by 0xleadwizard

Severity: Informational

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

Description: `SignatureCheckerLib.isValidERC6492SignatureNow` function won't work as expected if reverting verifier is not deployed, and there are some popular evm compatible chains where it is not deployed hence on those chains the `PublicERC6492Validator` if deployed won't work as expected:

```
/// @dev Returns whether `signature` is valid for `hash`.
/// If the signature is postfixed with the ERC6492 magic number, it will attempt
/// to use a reverting verifier to deploy / prepare the `signer` smart account
/// and do a `isValidSignature` check via the reverting verifier.
/// Note: This function is reentrancy safe.
/// The reverting verifier must be deployed.
/// Otherwise, the function will return false if `signer` is not yet deployed / prepared.
/// See: https://gist.github.com/Vectorized/846a474c855eee9e441506676800a9ad
function isValidERC6492SignatureNow(address signer, bytes32 hash, bytes memory signature)
    internal
    returns (bool isValid)
{
    /// @solidity memory-safe-assembly
    assembly {
        function callIsValidSignature(signer_, hash_, signature_) -> _isValid {
            let m_ := mload(0x40)
            let f_ := shl(224, 0x1626ba7e)
            mstore(m_, f_) // `bytes4(keccak256("isValidSignature(bytes32,bytes)"))`.
            mstore(add(m_, 0x04), hash_)
            let d_ := add(m_, 0x24)
            mstore(d_, 0x40) // The offset of the `signature` in the calldata.
            let n_ := add(0x20, mload(signature_))
            pop(staticcall(gas(), 4, signature_, n_, add(m_, 0x44), n_))
            _isValid :=
                and(
                    eq(mload(d_), f_),
                    staticcall(gas(), signer_, m_, add(returndatasize(), 0x44), d_, 0x20)
                )
        }
        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
            }
            if extcodesize(signer) {
                let o := add(signature, 0x20) // Signature bytes.
                isValid := callIsValidSignature(signer, hash, add(o, mload(add(o, 0x40))))
                if isValid { break }
            }
            let m := mload(0x40)
            mstore(m, signer)
            mstore(add(m, 0x20), hash)
            let willBeZeroIfRevertingVerifierExists :=
                call(
                    gas(), // Remaining gas.
                    >>> 0x00007bd799e4A591FeA53f8A8a3E9f931626Ba7e, // Reverting verifier.
                    0, // Send zero ETH.
                    m, // Start of memory.
                    add(returndatasize(), 0x40), // Length of calldata in memory.
                    staticcall(gas(), 4, add(signature, 0x20), n, add(m, 0x40), n), // 1.
                    0x00 // Length of returndata to write.
                )
            isValid := gt(returndatasize(), willBeZeroIfRevertingVerifierExists)
            break
        }
    }
}
```

The reverting verifier should be deployed on `0x00007bd799e4A591FeA53f8A8a3E9f931626Ba7e`.

Chains where the reverting verifier is not deployed on:

- BNB.

- AVALANCHE.

Recommendation: Deploy the reverting verifier before deploying `PublicERC6492Validator`.

3.2.8 Accounts cannot be deployed with ETH using `isValidSignatureNowAllowSideEffects()`

Submitted by MiloTruck

Severity: Informational

Context: PublicERC6492Validator.sol#L22-L27, SignatureCheckerLib.sol#L333-L337

Description: To deploy an account and grant spend permissions in one transaction, users first provide a signature for the permissions. The backend then wraps the signature using ERC-6492 with a call to `CoinbaseSmartWalletFactory.createAccount()`, which creates the account before verifying the signature.

This is done with Solady's `SignatureCheckerLib.isValidERC6492SignatureNowAllowSideEffects()`. However, calls with `isValidERC6492SignatureNowAllowSideEffects()` do not transfer value, as seen below:

```
if iszero(extcodesize(signer)) {
    if iszero(call(gas(), mload(o), 0, add(d, 0x20), mload(d), codesize(), 0x00)) {
        break
    }
}
```

Therefore, unlike transactions where `createAccount()` is called directly, accounts cannot be created with value when they are created with spend permissions.

Recommendation: Document this limitation.

3.2.9 Minor code improvements

Submitted by MiloTruck

Severity: Informational

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

Description/Recommendation:

1. SpendPermissionManager.sol#L81-L90 - `PERMISSION_TYPEHASH`, `PERMISSION_BATCH_TYPEHASH` and `PERMISSION_DETAILS_TYPEHASH` are missing their visibility specifiers:

```
  - bytes32 constant PERMISSION_TYPEHASH = keccak256(
  + bytes32 public constant PERMISSION_TYPEHASH = keccak256(
      "SpendPermission(address account,address spender,address token,uint160 allowance,uint48
↪   period,uint48 start,uint48 end,uint256 salt,bytes extraData)"
  );

  - bytes32 constant PERMISSION_BATCH_TYPEHASH = keccak256(
  + bytes32 public constant PERMISSION_BATCH_TYPEHASH = keccak256(
      "SpendPermissionBatch(address account,uint48 period,uint48 start,uint48 end,PermissionDetails[]
↪   permissions)PermissionDetails(address spender,address token,uint160 allowance,uint256 salt,bytes
↪   extraData)"
  );

  - bytes32 constant PERMISSION_DETAILS_TYPEHASH =
  + bytes32 public constant PERMISSION_DETAILS_TYPEHASH =
      keccak256("PermissionDetails(address spender,address token,uint160 allowance,uint256 salt,bytes
↪   extraData)");
```

2. SpendPermissionManager.sol#L393-L394 - The `uint256` cast below is unnecessary as `currentTimestamp` and `lastUpdatedPeriod.end` are both `uint48`:

```
  // last period still active if current timestamp within [start, end - 1] range.
  - bool lastPeriodStillActive = currentTimestamp < uint256(lastUpdatedPeriod.end);
  + bool lastPeriodStillActive = currentTimestamp < lastUpdatedPeriod.end;
```

3. SpendPermissionManager.sol#L457-L463 - The `totalSpend > type(uint160).max` check is redundant as it is implicitly checked by `totalSpend > spendPermission.allowance`:

```
- // check total spend value does not overflow max value
- if (totalSpend > type(uint160).max) revert SpendValueOverflow(totalSpend);

// check total spend value does not exceed spend permission
if (totalSpend > spendPermission.allowance) {
    revert ExceededSpendPermission(totalSpend, spendPermission.allowance);
}
```

3.2.10 Revert while revoking already expired permissions

Submitted by Sujith Somraaj

Severity: Informational

Context: SpendPermissionManager.sol#L287

Description: The `revoke` function revokes arbitrary spend permissions without validating/double-checking the input parameters.

Logically, it is good to revert if the current `block.timestamp` is beyond `spendPermission.end`, as the permission will have expired, and it makes less sense to revoke it.

The primary impact here would be the emission of `SpendPermissionRevoked` event for an already expired spend permission, which could mess up off-chain watchers.

Recommendation: Consider adding a revert to avoid revoking already expired spend permissions.

```
  function revoke(SpendPermission calldata spendPermission) external requireSender(spendPermission.account) {
+   if(spendPermission.end < block.timestamp) revert PermissionAlreadyExpired();
    // ...
  }
```

3.2.11 Ambiguity in Approval Flow Due to Lack of Revocation Status Check

Submitted by Kasheeda

Severity: Informational

Context: SpendPermissionManager.sol#L367

Description: This report identifies a potential ambiguity in the `SpendPermissionManager` contract's handling of approved and revoked permissions. Specifically, the `isApproved` function returns false for both unapproved and revoked permissions, making it difficult for an external application (app) to distinguish between these states. This could lead to accidental re-approval of revoked permissions if the app inadvertently reuses a salt, causing unintended transaction reverts and a confusing user experience.

The `SpendPermission` and `SpendPermissionBatch` structs are used to define specific token spending rights for a smart wallet account's owners or designated third-party spenders. These permissions include parameters such as the maximum allowance, period, validity start and end timestamps, and an arbitrary salt to ensure uniqueness.

When an app requests approval for a `SpendPermission`, it submits a signed permission to either `approveWithSignature` or `approveBatchWithSignature`. The contract then validates the signature and processes the approval. However, if a permission was previously revoked, the `isApproved` function returns false without distinguishing between an unapproved and a revoked permission. This ambiguity can lead to unintended reapproval requests.

For example, if an app reuses the same salt (either intentionally or accidentally), it may unknowingly request approval on a revoked permission. Once approved, even if it by the account (smart wallet) itself, any attempt to use this permission will fail in the spend function, where the following check will revert:

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

Since `isApproved` does not differentiate between revocation and lack of prior approval, the app cannot verify whether the permission was revoked before requesting reapproval. This limitation results in potential confusion and inefficiency in managing spend permissions, as the user's interaction with the app may fail with an `UnauthorizedSpendPermission` error. This error does not provide the necessary context—that the permission was revoked in the past—resulting in confusion and inefficiency in the permission management flow.

In the SpendPermissionManager contract, permissions can be approved or revoked, and approved permissions remain valid until the specified expiration (end timestamp). However, if a permission is revoked, the revocation status is recorded in _isRevoked mapping, and isApproved checks this mapping to determine whether the permission is revoked. The ambiguity arises because:

1. isApproved will return false if a permission is either unapproved or revoked, leading to a lack of clarity for the app.
2. The absence of an isRevoked function makes it challenging for apps to verify if a particular permission has been actively revoked or simply not yet approved.

Here's an example where this ambiguity might cause issues:

```
// The app sends a SpendPermission for approval
function requestApproval(SpendPermission calldata permission, bytes calldata signature) external {
    if (!spendPermissionManager.isApproved(permission)) {
        // If permission is not approved, request approval
        spendPermissionManager.approveWithSignature(permission, signature);
    }
}
```

If the app reuses a previously revoked permission's salt, it will end up requesting approval for a permission that was intended to remain permanently revoked. Without an isRevoked check, the app cannot differentiate between a new request and a revoked one, potentially causing a revert when the permission is reused.

In real-world applications, it's feasible that an app might accidentally reuse salts under certain conditions, like session resumption after network interruptions, caching issues, or unintended retries. Such scenarios could lead to unwanted reverts and complicate the signing flow, causing confusion for users who may believe they are approving a new permission rather than reactivating a previously revoked one.

Recommendation: To address this ambiguity, we recommend implementing an isRevoked function to allow external applications to explicitly check if a SpendPermission has been revoked:

```
/// @notice Checks if a spend permission has been explicitly revoked.
///
/// @param spendPermission The spend permission to check.
///
/// @return True if the permission has been revoked, false otherwise.
function isRevoked(SpendPermission calldata spendPermission) external view returns (bool) {
    bytes32 hash = getHash(spendPermission);
    return _isRevoked[hash][spendPermission.account];
}
```

By adding this function, apps can perform a more accurate check before requesting approval, thus improving reliability and user experience.