Base paymaster

Security Review

Cantina Managed review by:

Denis Milicevic, Lead Security Researcher

Chris Smith, Security Researcher

December 5, 2023

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

Base is a secure and low-cost Ethereum layer-2 solution built to scale the user base on-chain. The `paymaster` protocol contains a verifying paymaster contract that can be used for gas subsidies for ERC-4337 transactions. It contains a clone of the eth-infinitism VerifyingPaymaster with an additional `receive()` function for simple deposits.

From Oct 25th to Oct 27th the Cantina team conducted a review of Base paymaster on commit hash 08612ce4. The team identified a total of **25** issues in the following risk categories:

- Critical Risk: 0
- High Risk: 0
- Medium Risk: 0
- Low Risk: 7
- Gas Optimizations: 4
- Informational: 14

3 Findings

3.1 Low Risk

3.1.1 `Paymaster` has no recovery logic for force-sent ETH or tokens

Severity: Low Risk

Context: Paymaster.sol

Description: The contract could receive ETH via various force-send mechanisms, and would carry a balance which it isn't intended to do. They would stay stuck in the contract and be unrecoverable, although force-sends rarely occur in practice unless they result in some malicious effect, and no malicious effects were noted during the audit.

Another scenario is that users could accidently send wrapped ETH in token form in hopes of increasing the Paymaster's deposit balance. This is likelier to happen in practice and the underlying value would be lost.

Recommendation: The force-sent ETH could easily be handled by having the `entryPoint.depositTo` call pass `address(this).balance` instead of `msg.value` for the `value` parameter.

To handle accidently sent token scenarios, an additional `onlyOwner` protected function would be necessary to approve the owner or some other address to handle those tokens. This would only support ERC-20 based tokens, and it would realistically be impossible to support every specification.

Coinbase: Fixed with PR 17. We opted against making any changes for mistaken ERC-20 sends.

Cantina Managed: Fixed.

3.1.2 [Dependency] `entryPoint.withdrawTo` allows for re-entrancy

Severity: Low Risk

Context: StakeManager.sol#L121

Description: Due to lack of any re-entrancy guards, a `withdrawAddress` with appropriate privileges across depositors, could trigger additional withdrawals via re-entrancy from the same depositor or different depositors, again assuming they have the appropriate privileges to initiate a withdrawal from the depositor. The `withdrawalAddress` could also be a depositor themselves.

The accounting within the noted context is currently done correctly, whereby the balance is appropriately updated prior to the external call that opens up re-entrancy, thereby mitigating a DAO hack type of attack.

Recommendation: Although there is no obvious DAO type of attack possible due to the currently correct accounting, it is best practice to avoid leaving re-entrancy vectors open to untrusted actors, and implement appropriate re-entrancy guards.

Coinbase: Given this is a third-party dependency, we're avoiding fixing this for now, but may open an upstream PR for comment in the future.

Cantina Managed: Acknowledged.

3.1.3 Add sanity check in constructor that `_entryPoint` is contract

Severity: Low Risk

Context: BasePaymaster.sol#L21-L23

Description: The setting of the `entryPoint` variable currently lacks any validation. It could be set to the null address or an EOA while the `Paymaster` contract gets successfully deployed.

Recommendation: Adding a sanity check that the `_entryPoint` parameter passed to the constructor indeed points to a contract, would remove the aforementioned concerns. This can be done via a simple addition of:

```
require(address(_entryPoint).code.length > 0, "Passed _entryPoint is not currently a contract");
```

to ideally the `BasePaymaster` in the upstream dependency or within the `Paymaster` contract's constructor itself.

Additionally, OZ's `Address` library could be utilized which implements an `isContract()` function which works similar to the above.

Coinbase: Fixed in PR 14.

Cantina Managed: Fixed.

3.1.4 `Paymaster` will unexpectedly revert on 64-byte ECDSA short-sigs

Severity: Low Risk

Context: Paymaster.sol#L74-L80, ECDSA.sol#L56

Description: The Paymaster contract appears to signal intent for supporting both 64-byte and 65-byte signatures, as is apparent with the require statement which supports both lengths being passed. It specifically includes the require, so any incorrectly sized or unsupported signatures are reverted on before the OZ ECDSA library, so that the revert reason will be prefixed with `Paymaster:` rather than `ECDSA:` likely for improved error messages within the entrypoint contract for easier debugging.

The OZ library version used only supports 65-byte long signatures in the way it is implemented in the code here, and will revert on any other signatures that are not 65-bytes long when `ECDSA.recover(bytes32, bytes memory)` is called on ECDSA.sol#L56. The code incorrectly presumes support for both when passing a bytes signature, which used to be the case prior to OZ version 4.7.3. That version was patched to fix a high severity bug GHSA-4h98-2769-gh6h which lead to additional malleability vectors through the introduction of compact sigs.

The OZ library version used appears to be 4.9.3 @ fd81a96f01cc42ef1c9a5399364968d0e07e9e90 commit hash. This version contains the aforementioned malleability fix and drops the 64-bytes support the `Paymaster` implementations expected for that purpose.

Recommendation: For minimal changes, the contract should only allow 65 bytes length signatures, and appropriately update the require statement and comments to signal this intent.

If wishing to be gas-efficient on-chain long-term, 64 bytes signatures could be solely supported and just appropriately packed from the calldata. To support short-sigs, as of 4.7.3 OZ they must be explicitly called and supported via `recover(bytes32 hash, bytes32 r, bytes32 vs)`.

If wishing to support both, then the malleability concerns need to be addressed appropriately, by ensuring that message contents are tracked, consumed and thereby ensured to not be replayed via malleability. This seems an unnecessary complexity, considering the signer in this case will likely be a single entity, and does not need the flexbility of supporting both.

Coinbase: Fixed with PR 13.

Cantina Managed: Fixed.

3.1.5 Vanilla `renounceOwnership` from OZ `Ownable` could lead to unclaimable deposit

Severity: Low Risk

Context: Ownable.sol#L61-L63

Description: For safety, the `Ownable` library by OpenZeppelin has appropriate sanity checks in place, for the `transferOwnership` function to not allow changing the owner to `address(0)`, which would effectively lead to anyone losing ownership access controls on the contract. The function `renounceOwnership()` allows this and is intended to signal a wanton renouncing of such controls.

This can lead to a scenario where deposits are made by the original owner or on behalf, we can assume an ether balance of 10. If the owner were to call `renounceOwnership()`, they would lose access control, and the accompanying ether would be unwithdrawable from the deposits and potentially lost forever.

Since this is a `Paymaster` contract for ERC4337, assuming the contract and entire architecture is working correctly, the balance should technically still be available for consumption as long as a `verifyingSigner` is signing off on transactions to this paymaster.

Recommendation: Consider the need for a `renounceOwnership()` function in the contract model. If it is deemed unnecessary, and no scenario is intended whereby access controls should be forfeited, it is best to override it with a revert.

If it's deemed necessary, it would be most ideal to override `renounceOwnership()` in such a way that it attempts to pull the reported deposits from the entrypoint, via a preceding `entryPoint.withdrawTo(owner(), getDeposit());` call with appropriate checks on the external call's success. It would be a good idea to still keep the current simple iteration of `renounceOwnership()` but to reimplement it as `unsafeRenounceOwnership()` and only call it as a last resort in case the former fails, and assumes and outlines necessary recovery actions that should be manually taken by an owner prior to calling it.

Coinbase: Disabled ownership renouncement in PR 11.

Cantina Managed: Fixed.

3.1.6 No validation on `verifyingSigner` in Constructor

Severity: Low Risk

Context: Paymaster.sol#L29

Description: The `verifyingSigner` is fully trusted by the contract, but there is no verification of the address used in the `constructor`. The contract could be deployed with `address(0)` preventing it from functioning or, worse, the same address that deploys contract (and becomes `owner`) could be the `verifyingSigner`. This would prevent any recovery of the ETH placed in the contract if the signer ever became compromised.

Recommendation: Adding `require(_verifyingSigner != address(0))` would prevent accidentally deploying a bricked contract.

More importantly, given the two actor authorization system it would be best to add `require(_verifyingSigner != msg.sender)`. `msg.sender` will be used by `Ownable` as the `owner` of the contract. The `owner` is the only address that can withdraw the deposited ETH. Since the `verifyingSigner` is fully trusted to approve transactions spending that ETH and will need to be a relatively "hot" wallet, it would be safest to ensure that the `owner` is always a separate actor that can recover the ETH should the `verifyingSigner` ever become compromised. This can be done in the (new) `setVerifyingSigner` function, as well in an `override` for `transferOwnership`:

```
function transferOwnership(address newOwner) public override onlyOwner {
    require(newOwner != address(0), "Ownable: new owner is the zero address");
    require(newOwner != verifyingSigner, "Ownable: new owner is the verifyingSigner");
    _transferOwnership(newOwner);
}
```

Coinbase: Fixed in PR 5.

Cantina Managed: I think it would be good to add these checks to the new function too.

Coinbase: Good call, done in PR 18. Added the checks to `transferOwnership` too in PR 19.

Cantina Managed: Fixed.

3.1.7 `verifyingSigner` is immutable and fully trusted

Severity: Low Risk

Context: Paymaster.sol#L23

Description: By making this variable `immutable`, any change to the `verifyingSigner` address will require deploying a new `Paymaster`. This will require several steps including withdrawing and transferring the ETH that is in the current Paymaster. If the `verifyingSigner` becomes untrusted for some reason, this may require a quick response from the Coinbase team to protect the ETH from being used for bogus transactions.

Further, if the `verifyingSigner` becomes untrustworthy for any reason, there is no way to shut off the Paymaster contract (even removing all the ETH would still allow someone to deposit more and use the contract). Functionally this is not any different from them deploying their own `Paymaster` clone, funding it and signing transactions **except** that in this scenario, the Coinbase Paymaster that becomes compromised might have integrations that were already built into it.

Recommendation: Minimally, proper protections should be in place to secure the signing keys of the `verifyingSigner` address since it is completely trusted by this contract and losing control of that would allow a malicious actor to use.

Consider adding the ability to update the `verifyingSigner` address. Since the contract already is `Ownable` the permission structure is in place to add this. It would require some analysis of how to manage the switch so that there were not any valid transactions in flight that would become invalid when the address changed.

Another option would be to add the ability to shut off the contract so that no new deposits could be made and no transactions would be approved. This would kill the contract if the `verifyingSigner` is compromised while allowing the `owner` to withdraw the ETH.

Coinbase: Added the ability to update the `verifyingSigner` address in PR 4.

Cantina Managed: Fixed.

3.2 Gas Optimization

3.2.1 [Dependency] Call into `EntryPoint`'s receive fallback via `deposit()` for gas savings

Severity: Gas Optimization

Context: BasePaymaster.sol#L60-L65, StakeManager.sol#L35-L37

Description: The eth-infinitism contracts appear to strive for a high degree of gas efficiency. `BasePaymaster.deposit()` currently calls `entryPoint.depositTo...` directly along with accompanying arguments to successfully complete the call. The `entryPoint` also appears to have an implemented `receive` fallback function from its `StakeManager` import which would achieve the same effect.

Recommendation: Refactor the `deposit` to call into the `receive` fallback instead via a call that includes value to save a few hundred gas.

```diff
- entryPoint.depositTo{value : msg.value}(address(this));
+ payable(address(entryPoint)).call{value: msg.value}("");
```

Coinbase: Fixed in PR 16. We didn't modify the `BasePaymaster`, but updated our `receive` function.

Cantina Managed: Fixed.

3.2.2 `parsePaymasterAndData` function can be set to internal to save gas

Severity: Gas Optimization

Context: Paymaster.sol#L39-L40, Paymaster.sol#L89

Description: The `parsePaymasterAndData` function is only utilized from within the contract itself, although it is set with a public visibility specifier. It's a pure function and does some elementary operations that could easily be reimplemented wherever necessary outside of the contract's context. It is also not a part of the `IPaymaster` interface. Having functions set as public, when only used internally unnecessarily bloats the initial jump table of the contract, introducing overhead across the contract's function calls.

Similar conditions apply to the function `getHash`, albeit it does access `block.chainid` and may be utilized externally by an off-chain signer to get the hash to sign for a `userOp` via a local invocation of the function (`eth_call`).

Recommendation: Maximal gas savings would be gained by setting both to internal. There may be a usecase for `getHash` to stay public, and if it is applicable, it should be kept public. Setting just `parsePaymasterAndData` internal would yield gas savings as well and improve code clarity and intent.

Coinbase: Fixed with PR 12.

Cantina Managed: Fixed.

3.2.3 Use more efficient `calldataKeccak` for hashing calldata parameters

Severity: Gas Optimization

Context: Paymaster.sol#L47-48

Description: The eth-infinitism dependencies expose a `calldataKeccak` alternative which uses substantially less gas for calldata params than the built-in `keccak256`.

Recommendation: Utilize `calldataKeccak` instead on these lines for decent gas savings. The currently audited version meets output equivalency to the built-in, however, it would be wise for the client to include their own tests of `calldataKeccak`, including fuzzing with a high degree of runs, to ensure remaining equivalency across future versions.

Coinbase: Fixed in PR 10. Added a test for the hashing function, but no fuzzing added.

Cantina Managed: I've done 100M+ fuzzing runs so quite confident the version as it is, is good, but it's a good idea to have your own, in case the upstream code were to change.

A foundry fuzz test example to ensure equality with the built-in

```
/// forge-config: default.fuzz.runs = 10000
function testFuzz_calldataKeccak_equivalance(bytes calldata _x) public pure {
    assert(calldataKeccak(_x) == keccak256(_x));
}
```

Considered fixed.

3.2.4 `getHash` code can be simplified

Severity: Gas Optimization

Context: Paymaster.sol#L42-L45

Description: This function can be simplified by not creating the `sender` variable and invoking `userOp.getSender()` in the first parameter of the encoding.

Based on testing a similar function, this change would result in a very small bit of gas savings.

Recommendation: Consider applying the following change:

```
- address sender = userOp.getSender();
  return keccak256(
      abi.encode(
-         sender,
+         userOp.getSender(),
          userOp.nonce,
          keccak256(userOp.initCode),
          keccak256(userOp.callData),
          userOp.callGasLimit,
          userOp.verificationGasLimit,
          userOp.preVerificationGas,
          userOp.maxFeePerGas,
          userOp.maxPriorityFeePerGas,
          block.chainid,
          address(this),
          validUntil,
          validAfter
      )
  );
```

Before introducing `pack` the `VerifyingPaymaster` followed this pattern.

Coinbase: Fixed in PR 6.

Cantina Managed: Fixed.

3.3 Informational

3.3.1 `Paymaster` has no internal replay protection mechanisms

Severity: Informational

Context: Paymaster.sol

Description: The Paymaster contract validates that a signature indeed resolves to the `verifyingSigner` state variable, but it does not internally track for any replay protection. It is thus dependent on its associated `entryPoint` contract to appropriately track replays and appropriately filter them out.

This design greatly simplifies the `Paymaster` and also instead of making a `verifyingSigner` reliant on a stateful `Paymaster`, it allows a `Paymaster` contract to be more ephemeral, and instead ties a `verifyingSigner` to the initially associated `entryPoint`.

Recommendation: Since the `entryPoint` was out of scope within this audit, some assumptions are made here, but if the design is correctly understood, it is highly advisable to never re-use a `verifyingSigner` on a different or new `entryPoint` than the one that was initially set. Keys should be rotated for any new `entryPoint`, otherwise replays of prior ops may be possible.

Coinbase: Noted, we'll ensure to use a different `verifyingSigner` for paymasters on the same chain.

One alternative might be to include the paymaster contract address in the hash that is signed, to avoid being able to use the same signature on different paymasters.

Cantina Managed: Acknowledged.

3.3.2 Include all inherited constructor calls in derived contracts

Severity: Informational

Context: Paymaster.sol#L28

Description: Contracts with deep inheritance chains may have multiple constructor calls that are non-obvious and implicitly called on deployment.

Recommendation: For code clarity, explictly denote all inherited constructors which communicates to end-users that the devs of the derived contract understand all the underlying dependency actions occurring and also makes it more obvious to contract readers without having to dig into the inheritance chain. As a note, their ordering within the constructor of the most derived contract does not affect their actual call order, which is derived by inheritance order.

```diff
- constructor(IEntryPoint _entryPoint, address _verifyingSigner) BasePaymaster(_entryPoint) {
+ constructor(IEntryPoint _entryPoint, address _verifyingSigner) BasePaymaster(_entryPoint) Ownable() {
```

Coinbase: Fixed in PR 15.

Cantina Managed: Fixed.

3.3.3 [Dependency] `UserOperationLib` exposes `min` function unintended for `UserOperation`

Severity: Informational

Context: UserOperation.sol#L88-L90

Description: The `UserOperationLib` library implements a function intended for `uint256` and likely only internal library use, but exposes it due to being declared with an internal visiblity specifier. The library in question is intended to be used upon `UserOperation` type structs.

Recommendation: It is best to unexpose this function by setting it to `private`, as it is likely unintended to be piggybacked for use outside itself, and if a `min` function is needed a developer can utilize a more appropriate and focused `uint256` Math lib.

Coinbase: Given this is a third-party dependency, we're avoiding fixing this for now, but may open an upstream PR for comment in the future.

Cantina Managed: Acknowledged.

3.3.4 [Dependency] Avoid `uint` declarations and opt for `uint256` for clarity

Severity: Informational

Context: Helpers.sol#L23

Description: The function parameter is declared as a `uint` type which is an alias for `uint256`.

Recommendation: It is best practice to use `uint256` for full clarity and most importantly to stay consistent on usage.

Coinbase: Given this is a third-party dependency, we're avoiding fixing this for now, but may open an upstream PR for comment in the future.

Cantina Managed: Acknowledged.

3.3.5 `validUntil` values of 0 signal infinity and are transformed into `type(uint48).max`

Severity: Informational

Context: Helpers.sol#L22-L27

Description: The `validUntil` variable is normally a 48-bit timestamp in UNIX seconds to signify up to a point in time a UserOp is valid for. There is an edge case, where setting 0, assumes an infinite validity, and in the code it essentially is tranformed to `(2**48) - 1`.

Recommendation: Appropriately account for this condition within the `verifyingSigner` off-chain verification logic.

Coinbase: Noted, we'll keep this in mind in our offchain logic.

Cantina Managed: Acknowlegded.

3.3.6 [Dependency] Overflow possible on user specified variable in unchecked block

Severity: Informational

Context: UserOperation.sol#L52-L59

Description: The line in question is

```
return min(maxFeePerGas, maxPriorityFeePerGas + block.basefee);
```

It is wrapped within an unchecked block and does arithmetic on a user settable variable. If we assume a basefee of 20 gwei (2×10^{10}), a user could potentially pass a value of `2**256 - 20 gwei + 1` to overflow that arithmetic operation to 1, and unexpectedly return a value lower than both `maxFeePerGas`and `maxPriorityFeePerGas` from this function without revert.

This can lead to higher severity issues, however, since this code is unused in the audited `Paymaster` implementation, it is noted here as informational. It is utilized in the sample `DeployingPaymaster` but its effects haven't been thoroughly explored as that contract isn't within the scope of this audit.

Recommendation: This unchecked block if there for gas saving purposes only provides negligible savings for a single arithmetic operation while exposing the function to overflow potential, and the unchecked block should be omitted to ensure it is appropriately checked for overflows.

Otherwise dependence on this function should be avoided, and any unchecked code within the eth-infinitism dependencies further explored for correctness that may have beeen outside this audit scope.

Coinbase: Given this is a third-party dependency, and our implementation is currently unaffected, we're avoiding fixing this for now, but may open an upstream PR for comment in the future.

Cantina Managed: Acknowledged.

3.3.7 [Dependency] Signed `UserOperation` struct has neighboring variable-length types

Severity: Informational

Context: UserOperation.sol#L8-L35

Description: The `UserOperation` struct from eth-infinitism's `account-abstraction` dependency is the payload used for signing. Contained within are 2 parameters, `initCode` and `callData` which are bytes types and in neighboring slots. When it comes to signatures, this can introduce additional malleability and replay concerns, in the case `abi.encodePacked` is utilized when hashing the struct, in that the individual contents of `initiCode` and `callData` could be altered, while keeping the contiguous data the same, and thereby allowing signature re-use across differing payloads.

In this case, the dependency and general codebase relies on `abi.encode` which does not pack the parameters and avoids these pitfalls.

Recommendation: Avoid using `abi.encodePacked` to not introduce aforementioned pitfalls which could lead to exploits. In addition, the upstream dependency could be made safer and less error prone by splitting the variable lengh types with a static length primitive type, so they do not occupy neighboring slots.

Coinbase: Given this is a third-party dependency, we're avoiding fixing this for now, but may open an upstream PR for comment in the future.

Cantina Managed: Acknowledged,

3.3.8 Unused `using Lib for` statement

Severity: Informational

Context: Paymaster.sol#L20

Description: There is a `using ECDSA for bytes32;` statement declared in the contract, however, it is not utilized by the contract for `bytes32`, as the library is explicitly called within the contract.

Recommendation: Either remove this unnecessary statement for minimal changes, and if wishing to simply rely on the explicit calls to the library or refactor the two library calls to actually utilize the statement. An example being Paymaster.sol#L77:

```diff
- if (verifyingSigner != ECDSA.recover(hash, signature)) {
+ if (verifyingSigner != hash.recover(signature)) {
```

Coinbase: Fixed in PR 9.

Cantina Managed: Fixed.

3.3.9 Avoid using dead code as workaround for unused function parameter

Severity: Informational

Context: Paymaster.sol#L71

Description: The line utilizes no-effects code as a workaround to hide a compiler warning regarding an unused function parameter. This is generally bad practice as it hurts readability and code clarity.

Recommendation: The appropriate handling of an unused function parameter is to remove the variable name, or ideally as done with the `userOpHash`, to comment out the name and not add unnecessary dead code.

Coinbase: Fixed in PR 8.

Cantina Managed: Fixed.

3.3.10 Function declaration formating consistency

Severity: Informational

Context: Paymaster.sol#L89

Description: Other long function declarations are broken into two lines.

Recommendation:

```
- function parsePaymasterAndData(bytes calldata paymasterAndData) public pure returns(uint48 validUntil,
↪  uint48 validAfter, bytes calldata signature) {
+ function parsePaymasterAndData(bytes calldata paymasterAndData)
+ public pure returns(uint48 validUntil, uint48 validAfter, bytes calldata signature) {
```

Coinbase: Fixed in PR 7.

Cantina Managed: Fixed.

3.3.11 `solhint-disable` rule no longer needed

Severity: Informational

Context: Paymaster.sol#L5

Description: With the removal of the `pack()` function, this solhint rule no longer needs to be disabled.

Recommendation: Remove this line to re-enable the rule.

```
  // SPDX-License-Identifier: GPL-3.0
  pragma solidity ^0.8.12;

  /* solhint-disable reason-string */
- /* solhint-disable no-inline-assembly */
```

Coinbase: Fixed in PR 3.

Cantina Managed: Fixed.

3.3.12 `pragma` version definition

Severity: Informational

Context: Paymaster.sol#L2

Description: It is generally considered best practice to avoid using floating Solidity version declarations.

Recommendation: Lock the pragma to the specific version you wish to use (in this case, `foundry.toml` defines `0.8.20`.

Coinbase: Fixed in PR 2.

Cantina Managed: FIxed.

3.3.13 Packed and Parsed Validation Data

Severity: Informational

Context: eth-infinitism/account-abstraction/blob/v0.6.0/contracts/core/Helpers.sol#L7-L67, Paymaster.sol#L56-L57, Paymaster.sol#L81-L92

Description: In the `Paymaster.sol` contract and in the `Helpers._packValidationData` functions, the timestamps for signature validity are handled in `validUntil` then `validAfter` order. This means that when unpacking this data, the first timestamp that is unpacked is the `validUntil` and the second one is `validAfter`. This is the inverse of how time ranges are usually presented (i.e. usually one would say "from this time to this later time").

Further, if the `Helpers._parseValidationData` or `Helpers._intersectTimeRange` functions are used, they will return a tuple where the data are `aggregator address, validAfter timestamp, validUntil timestamp` (reversing the order in which the timestamps were passed in and packed).

The `Paymaster.getHash` function uses the same `validUntil` and `validAfter` ordering as the Helper packing functions.

Recommendation: Since all these smart contract operations handled the data correctly (i.e. `parsing` properly assigns `validUntil` and `validAfter` before returning them in the other order), there is no issue within the code reviewed.

However, the inconsistent ordering and packing in the reverse order from how time ranges tend to be consumed could lead to confusion and incorrect integrations.

When parsing any packed Validation Data off-chain, it is important to note that the order of `validAfter` and `validUntil` are reversed from the way that they are packed and handled in the deployed `Paymaster.sol` contract.

Coinbase: Noted that we have to be careful to pack the `validAfter` and `validUntil` fields when preparing the offchain signature. We decided not to make any code changes here for now, given the dependency on the upstream repo.

Cantina Managed: Acknowledged.

3.3.14 Testing

Severity: Informational

Context: ./test

Description: The current repository does not contain tests. It is intentionally a very close clone of eth-infinitism/VerifyingPaymaster.sol However, there are a few code changes and additions (see diff image below). Further by moving `Paymaster.sol` to its own repo, its dependencies could diverge.

I have used `Paymaster.sol`'s code in eth-infinitism/VerifyingPaymaster.sol and run their tests and they all pass:

```
  EntryPoint with VerifyingPaymaster
    #parsePaymasterAndData
0xfe5f411481565fbf70d8d33d992c78196e014b9000000000000000000000000000000000000000000000000000000000deadbeef0000 ⌋
↪   0000000000000000000000000000000000000000000000000000000012341234
       should parse data properly
    #validatePaymasterUserOp
       should reject on no signature
       should reject on invalid signature
       succeed with valid signature
      with wrong signature
         should return signature error (no revert) on wrong signer signature
         handleOp revert on signature failure in handleOps
```

Recommendation: While this code currently passes the existing `eth-infinitism` tests for the `VerifyingPaymaster` we would recommend creating a local test suite that tests `Paymaster.sol` directly and tests its additional/changed functionality. This will ensure correctness now and help protect against possible future regressions. Porting the tests over to `forge` from `eth-infinitism`'s Hardhat tests would likely not be too difficult and could lay the foundation for future testing such as Foundry's invariant tests.

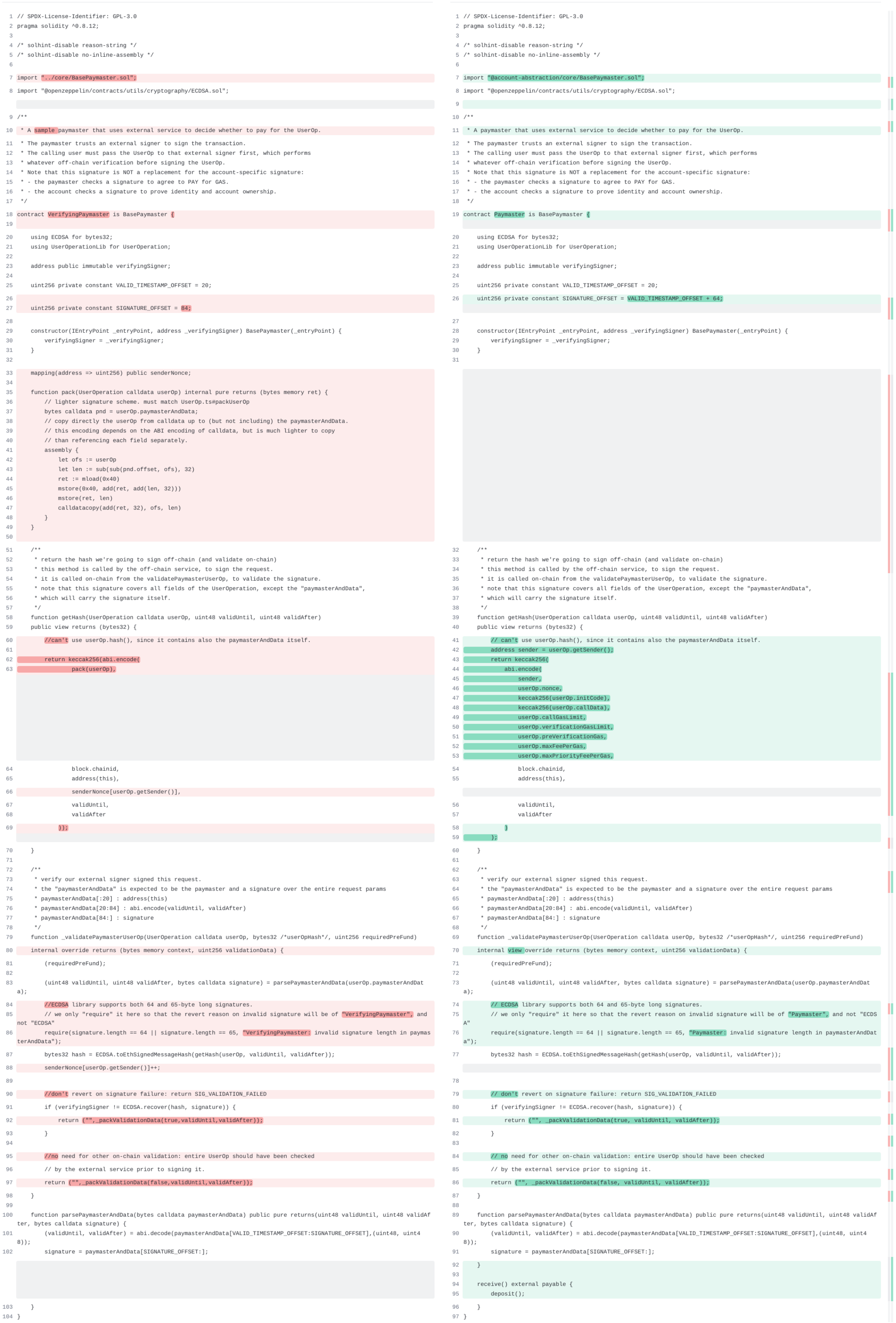
See contract diff check in Appendix.

Coinbase: Implemented in PR 1.

Cantina Managed: Reviewed tests from this PR and final version on `main`. They look good and do not omit any obvious paths.

3.4 Appendix

3.4.1 Contract Diff Check



```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.12;

/* solhint-disable reason-string */
/* solhint-disable no-inline-assembly */

import "../core/BasePaymaster.sol";
import "@openzeppelin/contracts/utils/cryptography/ECDSA.sol";
/**
 * A sample paymaster that uses external service to decide whether to pay for the UserOp.
 * The paymaster trusts an external signer to sign the transaction.
 * The calling user must pass the UserOp to that external signer first, which performs
 * whatever off-chain verification before signing the UserOp.
 * Note that this signature is NOT a replacement for the account-specific signature:
 * - the paymaster checks a signature to agree to PAY for GAS.
 * - the account checks a signature to prove identity and account ownership.
 */
contract VerifyingPaymaster is BasePaymaster {

    using ECDSA for bytes32;
    using UserOperationLib for UserOperation;

    address public immutable verifyingSigner;

    uint256 private constant VALID_TIMESTAMP_OFFSET = 20;

    uint256 private constant SIGNATURE_OFFSET = 84;

    constructor(IEntryPoint _entryPoint, address _verifyingSigner) BasePaymaster(_entryPoint) {
        verifyingSigner = _verifyingSigner;
    }

    mapping(address => uint256) public senderNonce;

    function pack(UserOperation calldata userOp) internal pure returns (bytes memory ret) {
        // lighter signature scheme. must match UserOp.ts#packUserOp
        bytes calldata pnd = userOp.paymasterAndData;
        // copy directly the userOp from calldata up to (but not including) the paymasterAndData.
        // this encoding depends on the ABI encoding of calldata, but is much lighter to copy
        // than referencing each field separately.
        assembly {
            let ofs := userOp
            let len := sub(sub(pnd.offset, ofs), 32)
            ret := mload(0x40)
            mstore(0x40, add(ret, add(len, 32)))
            mstore(ret, len)
            calldatacopy(add(ret, 32), ofs, len)
        }
    }

    /**
     * return the hash we're going to sign off-chain (and validate on-chain)
     * this method is called by the off-chain service, to sign the request.
     * it is called on-chain from the validatePaymasterUserOp, to validate the signature.
     * note that this signature covers all fields of the UserOperation, except the "paymasterAndData",
     * which will carry the signature itself.
     */
    function getHash(UserOperation calldata userOp, uint48 validUntil, uint48 validAfter)
    public view returns (bytes32) {
        //can't use userOp.hash(), since it contains also the paymasterAndData itself.

        return keccak256(abi.encode(
                pack(userOp),
                block.chainid,
                address(this),
                senderNonce[userOp.getSender()],
                validUntil,
                validAfter
            ));
    }

    /**
     * verify our external signer signed this request.
     * the "paymasterAndData" is expected to be the paymaster and a signature over the entire request params
     * paymasterAndData[:20] : address(this)
     * paymasterAndData[20:84] : abi.encode(validUntil, validAfter)
     * paymasterAndData[84:] : signature
     */
    function _validatePaymasterUserOp(UserOperation calldata userOp, bytes32 /*userOpHash*/, uint256 requiredPreFund)
    internal override returns (bytes memory context, uint256 validationData) {
        (requiredPreFund);

        (uint48 validUntil, uint48 validAfter, bytes calldata signature) = parsePaymasterAndData(userOp.paymasterAndData);
        //ECDSA library supports both 64 and 65-byte long signatures.
        // we only "require" it here so that the revert reason on invalid signature will be of "VerifyingPaymaster", and not "ECDSA"
        require(signature.length == 64 || signature.length == 65, "VerifyingPaymaster: invalid signature length in paymasterAndData");
        bytes32 hash = ECDSA.toEthSignedMessageHash(getHash(userOp, validUntil, validAfter));
        senderNonce[userOp.getSender()]++;

        //don't revert on signature failure: return SIG_VALIDATION_FAILED
        if (verifyingSigner != ECDSA.recover(hash, signature)) {
            return ("",_packValidationData(true,validUntil,validAfter));
        }

        //no need for other on-chain validation: entire UserOp should have been checked
        // by the external service prior to signing it.
        return ("",_packValidationData(false,validUntil,validAfter));
    }

    function parsePaymasterAndData(bytes calldata paymasterAndData) public pure returns(uint48 validUntil, uint48 validAfter, bytes calldata signature) {
        (validUntil, validAfter) = abi.decode(paymasterAndData[VALID_TIMESTAMP_OFFSET:SIGNATURE_OFFSET],(uint48, uint48));
        signature = paymasterAndData[SIGNATURE_OFFSET:];
    }
}
```

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.12;

/* solhint-disable reason-string */
/* solhint-disable no-inline-assembly */

import "@account-abstraction/core/BasePaymaster.sol";
import "@openzeppelin/contracts/utils/cryptography/ECDSA.sol";

/**
 * A paymaster that uses external service to decide whether to pay for the UserOp.
 * The paymaster trusts an external signer to sign the transaction.
 * The calling user must pass the UserOp to that external signer first, which performs
 * whatever off-chain verification before signing the UserOp.
 * Note that this signature is NOT a replacement for the account-specific signature:
 * - the paymaster checks a signature to agree to PAY for GAS.
 * - the account checks a signature to prove identity and account ownership.
 */
contract Paymaster is BasePaymaster {
    using ECDSA for bytes32;
    using UserOperationLib for UserOperation;

    address public immutable verifyingSigner;

    uint256 private constant VALID_TIMESTAMP_OFFSET = 20;
    uint256 private constant SIGNATURE_OFFSET = VALID_TIMESTAMP_OFFSET + 64;

    constructor(IEntryPoint _entryPoint, address _verifyingSigner) BasePaymaster(_entryPoint) {
        verifyingSigner = _verifyingSigner;
    }

    /**
     * return the hash we're going to sign off-chain (and validate on-chain)
     * this method is called by the off-chain service, to sign the request.
     * it is called on-chain from the validatePaymasterUserOp, to validate the signature.
     * note that this signature covers all fields of the UserOperation, except the "paymasterAndData",
     * which will carry the signature itself.
     */
    function getHash(UserOperation calldata userOp, uint48 validUntil, uint48 validAfter)
    public view returns (bytes32) {
        // can't use userOp.hash(), since it contains also the paymasterAndData itself.
        address sender = userOp.getSender();
        return keccak256(
            abi.encode(
                sender,
                userOp.nonce,
                keccak256(userOp.initCode),
                keccak256(userOp.callData),
                userOp.callGasLimit,
                userOp.verificationGasLimit,
                userOp.preVerificationGas,
                userOp.maxFeePerGas,
                userOp.maxPriorityFeePerGas,
                block.chainid,
                address(this),
                validUntil,
                validAfter
            )
        );
    }

    /**
     * verify our external signer signed this request.
     * the "paymasterAndData" is expected to be the paymaster and a signature over the entire request params
     * paymasterAndData[:20] : address(this)
     * paymasterAndData[20:84] : abi.encode(validUntil, validAfter)
     * paymasterAndData[84:] : signature
     */
    function _validatePaymasterUserOp(UserOperation calldata userOp, bytes32 /*userOpHash*/, uint256 requiredPreFund)
    internal view override returns (bytes memory context, uint256 validationData) {
        (requiredPreFund);

        (uint48 validUntil, uint48 validAfter, bytes calldata signature) = parsePaymasterAndData(userOp.paymasterAndData);
        // ECDSA library supports both 64 and 65-byte long signatures.
        // we only "require" it here so that the revert reason on invalid signature will be of "Paymaster", and not "ECDSA"
        require(signature.length == 64 || signature.length == 65, "Paymaster: invalid signature length in paymasterAndData");
        bytes32 hash = ECDSA.toEthSignedMessageHash(getHash(userOp, validUntil, validAfter));

        // don't revert on signature failure: return SIG_VALIDATION_FAILED
        if (verifyingSigner != ECDSA.recover(hash, signature)) {
            return ("", _packValidationData(true, validUntil, validAfter));
        }

        // no need for other on-chain validation: entire UserOp should have been checked
        // by the external service prior to signing it.
        return ("", _packValidationData(false, validUntil, validAfter));
    }

    function parsePaymasterAndData(bytes calldata paymasterAndData) public pure returns(uint48 validUntil, uint48 validAfter, bytes calldata signature) {
        (validUntil, validAfter) = abi.decode(paymasterAndData[VALID_TIMESTAMP_OFFSET:SIGNATURE_OFFSET],(uint48, uint48));
        signature = paymasterAndData[SIGNATURE_OFFSET:];
    }

    receive() external payable {
        deposit();
    }
}
```