# DeFi Wonderland OpUSDC

## Security Review


Cantina Managed review by:

**0xIcingdeath**, Lead Security Researcher

**Optimum**, Lead Security Researcher

**R0bert**, Security Researcher

**Shung**, Associate Security Researcher



August 13, 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

Wonderland is a group of developers, researchers, and data scientists with the mission to discover, partner with, and empower innovators to create open, permissionless, decentralized financial solutions.

From Jul 14th to Jul 19th the Cantina team conducted a review of defi-wonderland-opUSDC on commit hash b568e71d. After reviewing the issues raised by the researchers, the necessary fixes along with changes in the protocol logic were applied on commit hash eb625f95, for a second review on Jul 20th. The team identified a total of **21** issues in the following risk categories:

- Critical Risk: 0
- High Risk: 0
- Medium Risk: 3
- Low Risk: 5
- Gas Optimizations: 0
- Informational: 13

Finally, the Cantina team reviewed wonderland-opUSD changes holistically on commit hash 79f128ea208a90a2fdead67c7e2ac21cf057b6fd and determined that all issues were resolved and no new issues were identified.

# 3 Findings

## 3.1 Medium Risk

### 3.1.1 Temporary failed L1 to L2 token transfers might lock tokens in L1 if replayed after migrating to native USDC

**Severity:** Medium Risk

**Context:** L2OpUSDCBridgeAdapter.sol#L205

**Description:** The current system uses a well-known token bridging mechanism, locking tokens on one chain and minting them on the other. The `L2OpUSDCBridgeAdapter` contract is assigned the minter role within the bridged USDC contract. This role is intended to be revoked as part of the migration process as described in bridged_USDC_standard.md:

> Additionally, the partner is expected to remove all configured minters prior to (or concurrently with) transferring the roles to Circle.

Although unlikely, messages to Optimism L2 chains might fail due to business logic issues (pausing of the bridged USDC contract for instance) or an out-of-gas exception on L2 (as described in replaying-messages). In the current system, this could cause calls to `USDC.mint` to revert. If messages are not replayed before migrating the bridged USDC contract to native USDC, any call to `USDC.mint` will revert since the bridged USDC minting permission was revoked for the `L2OpUSDCBridgeAdapter` contract. This will result in user funds being locked inside the `L1OpUSDCBridgeAdapter` contract in L1.

**Impact:** High since users funds will be locked in L1.

**Likelihood:** Low since it is less likely that messages will have to be replayed and will be replayed only after the migration.

**Recommendation:** To mitigate this issue consider implementing the following changes in `L2OpUSDCBridgeAdapter`:

1. Change the `isMessagingDisabled` variable from a boolean to an enum with three values: `Active(0)`, `Paused(1)`, `Upgraded(2)`. In addition consider changing its name to `messengerStatus`.
2. Change `receiveMigrateToNative` to set the state of `messengerStatus` to `Upgraded` instead of setting `isMessagingDisabled` to `true`.
3. Change `receiveStopMessaging` and `receiveResumeMessaging` to turn the `messengerStatus` from Paused to Active respectively.
4. Move the logic of the `.sendMessage` call and event emission of `MessageSent` from the two different implementations of `sendMessage` to an internal function named `_sendMessage`.
5. Change the `receiveMessage` function so that in case the `messengerStatus` is `Upgraded`, then the call to `mint` will be wrapped in a try and catch clause where the catch block filters the cause for the failure and in case it is equal to "FiatToken: caller is not a minter" then call `_sendMessage` which will in turn call `receiveMessage` on L1 with `_user` and `_amount`.

*Please note that this proposed solution is not perfect. In the rare case of multiple bridged USDC minters, users' tokens that remain locked on L1 might be permanently burned if Circle calls `burnLockedUSDC`.*

**Wonderland:** Fixed in commit eb625f95 by sending a message back to L1 to withdraw to the original spender.

**Cantina Managed:** Fixed by implementing the auditor's recommendation with slight changes that achieve the same result.

### 3.1.2 BurnAmount sent from L2 to L1 may not be accurate as it does not account for pending failing messages

**Severity:** Medium Risk

**Context:** L2OpUSDCBridgeAdapter.sol#L81

**Description:** In order to follow Circle's Bridged USDC standard, the `L2OpUSDCBridgeAdapter` contract implements the function `receiveMigrateToNative()`:

```
/**
 * @notice Initiates the process to migrate the bridged USDC to native USDC
 * @dev Full migration cant finish until L1 receives the message for setting the burn amount
 * @param _roleCaller The address that will be allowed to transfer the USDC roles
 * @param _setBurnAmountMinGasLimit Minimum gas limit that the setBurnAmount message can be executed on L1
 */
function receiveMigrateToNative(address _roleCaller, uint32 _setBurnAmountMinGasLimit) external
↪  onlyLinkedAdapter {
  isMessagingDisabled = true;
  roleCaller = _roleCaller;

  uint256 _burnAmount = IUSDC(USDC).totalSupply();

  ICrossDomainMessenger(MESSENGER).sendMessage(
    LINKED_ADAPTER, abi.encodeWithSignature('setBurnAmount(uint256)', _burnAmount), _setBurnAmountMinGasLimit
  );

  emit MigratingToNative(MESSENGER, _roleCaller);
}
```

This function sends as the `_burnAmount` the current L2 USDC total supply. However, this is not totally compliant with Bridged USDC Standard as this property would not be respected:

*The `setBurnAmount()` function must burn the amount of USDC held by the bridge that corresponds **precisely to the circulating total supply of bridged USDC** established by the supply lock.*

This amount is not accurate as the total supply does not account for pending failed messages that are present at the time of the call in the L2Messenger contract. These messages could be called in order to mint new bridged USDC tokens as the `L2OpUSDCBridgeAdapter.receiveMessage()` function will be still working after the `L2OpUSDCBridgeAdapter.receiveMigrateToNative()` was triggered:

```
/**
 * @notice Receive the message from the other chain and mint the bridged representation for the user
 * @dev This function should only be called when receiving a message to mint the bridged representation
 * @param _user The user to mint the bridged representation for
 * @param _amount The amount of tokens to mint
 */
function receiveMessage(address _user, uint256 _amount) external override onlyLinkedAdapter {
  // Mint the tokens to the user
  IUSDC(USDC).mint(_user, _amount);
  emit MessageReceived(_user, _amount, MESSENGER);
}
```

**Impact:** Medium, as the Circle's Bridged USDC standard is not fully respected and the `_burnAmount` is not accurate.

**Likelihood:** High, a single failed message stuck in the L2Messenger by the time of the `receiveMigrateToNative()` will trigger this issue.

**Recommendation:** Record any `L2OpUSDCBridgeAdapter.receiveMessage()` amounts after the `receiveMigrateToNative()` has been triggered. Then, this extra amount could be burned any time with a separate function in `L2OpUSDCBridgeAdapter` which would call the corresponding function in the `L1OpUSDCBridgeAdapter` with the extra amount.

**Wonderland:** Fixed in commit eb625f95.

**Cantina Managed:** Fix verified. The fix implemented consists in sending a message back to L1 to withdraw to the original spender. This was achieved with the try/catch block added in the L1 adapter to the `receiveMessage()` function and through the addition of the `receiveWithdrawBlacklistedFundsPostMigration()` function.

Any funds that will be sent from the L2 adapter after a migration (L2 adapter status = Deprecated), will either be sent directly to the user in the L1 adapter or registered in the `blacklistedFundsDetails` mapping for a future claim in case of a failure.

### 3.1.3 USDC sent may be permanently locked/burnt in/by the Adapter if the `_to` address is blacklisted in the destination chain

**Severity:** Medium Risk

**Context:** L1OpUSDCBridgeAdapter.sol#L194, L1OpUSDCBridgeAdapter.sol#L218, L2OpUSDCBridgeAdapter.sol#L138, L2OpUSDCBridgeAdapter.sol#L164, USDC current implementation

**Description:** The current USDC contract implements a blacklist through the `notBlacklisted(address)` modifier which only allows transferring/minting USDC from/to addresses that are not blacklisted:

```
/**
 * @dev Throws if argument account is blacklisted.
 * @param _account The address to check.
 */
modifier notBlacklisted(address _account) {
    require(
        !_isBlacklisted(_account),
        "Blacklistable: account is blacklisted"
    );
    _;
}
```

If the `_to` address set in the `sendMessage()` call is a blacklisted address in the USDC contract of the destination chain the transfer/minting will revert and the message will be permanently stuck in a failed state. Let's imagine the following scenario:

1. Alice has 10000 USDC in Optimism and wants to bridge them to Bob in the Ethereum mainnet.
2. Alice checks if Bob is blacklisted in the Ethereum mainnet. Bob is not.
3. Alice calls `L2OpUSDCBridgeAdapter.sendMessage(BOB, 10000e6, 2000000)`.
4. As per the Optimism docs:

   Transactions sent from L2 to L1 take approximately 7 days to get from OP Mainnet to Ethereum"
5. Before these 7 days have passed, Bob is blacklisted in the USDC Ethereum Mainnet contract.
6. When the `relayMessage()` function is called it reverts/fails as Bob was blacklisted in Ethereum mainnet.
7. Alice's tokens were already burnt in the `L2OpUSDCBridgeAdapter` without any possible way to recover them.

Note, that this issue is more likely to occur in L2 $\Rightarrow$ L1 than L1 $\Rightarrow$ L2 operations, as the transactions sent from L1 to L2 take approximately 1-3 minutes to get from Ethereum to the L2, while transactions sent from L2 to L1 can take around 7 days.

**Impact:** High as USDC may be permanently locked/burnt in/by the Adapter contracts.

**Likelihood:** Medium as an account might not be blacklisted when the bridge transaction is initiated, but it could be blacklisted by the time the transaction is finalized.

**Recommendation:** Consider transferring the USDC tokens to the `_to` address within a try/catch block in both L1/L2 Adapters. If the transfer fails we can assume that the destination address is blacklisted so the `amount` should be stored in an internal mapping to keep track of all the "stuck" funds. An `onlyOwner` function should also be implemented to handle these funds in a centralized manner.

**Wonderland:** Fixed in commit eb625f95.

**Cantina Managed:** Fix verified. The fix involved adding a `try/catch` code block as recommended to handle the transfer of USDC funds. The USDC remains in the adapter and a function `withdrawBlacklistedFunds()` was added which allows the user to withdraw if they are removed from the blacklist.

On the L2 adapter the same approach of using a try/catch block was implemented to handle the USDC mints.

If `receiveMessage()` is called in the L2 adapter after the migration a message will be sent back to L1 to withdraw to the original spender.

In addition, the project decided to revert cross-chain USDC transactions where the beneficiary is black-listed on the source chain.

## 3.2 Low Risk

### 3.2.1 `L1OpUSDCBridgeAdapter`,`L2OpUSDCBridgeAdapter`: Spec mismatch, contracts should be upgradeable

**Severity:** Low Risk

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

**Description:** According to Circle's documentation, the two bridge adapter contracts should be upgradeable; however, the current implementation does not reflect this.

**Recommendation:** Consider changing these contracts to be upgradeable, and if not, consider asking Circle to change the requirement to be optional instead.

**Wonderland:** Fixed in commit eb625f95.

**Cantina Managed:** In OpUSDCBridgeAdapter.sol#L29-L39, the storage `__gap` should be declared after all the other storage variables (for context, see Openzeppelin's documentation on upgradeable contracts).

**Wonderland:** Fixed in commit 79f128ea.

**Cantina Managed:** Fixed by implementing the auditor's recommendation.

### 3.2.2 `L1OpUSDCFactory.deploy`: Missing event emissions for `_l2Factory` and `_l2Adapter`

**Severity:** Low Risk

**Context:** L1OpUSDCFactory.sol#L67

**Description:** During the execution of `L1OpUSDCFactory.deploy`, three different contracts are deployed: `L1OpUSDCBridgeAdapter`, `L2OpUSDCBridgeAdapter`, and `L2OpUSDCFactory`. The addresses of these deployed contracts are returned as the function's return values. However, since this function is intended to be called by an externally owned account (EOA), retrieving these addresses becomes difficult because return values cannot be fetched by the calling EOA. While the function emits an event with the address of the newly deployed `L1OpUSDCBridgeAdapter` contract, it does not emit events for the other two contracts. The absence of event emissions for the other two contracts could, in the worst-case scenario, lead the caller to pick incorrect addresses. These addresses might be manipulated by a front-runner who calls the `deploy` function.

**Recommendation:** Consider adding two event emissions for these addresses and make sure the caller logic uses the values of these events for future interactions.

**Wonderland:** Fixed in commit eb625f95.

**Cantina Managed:** Fixed by implementing the auditor's recommendation. Given that the contract has been renamed to deploy instead of factory, consider renaming the `_l2FactoryInitCode` variable (CrossChainDeployments.sol#L39) to `_l2DeployInitCode` for consistency.

**Wonderland:** Fixed in commit 79f128ea.

**Cantina Managed:** Fix verified.

### 3.2.3 Bridging tokens to `address(0)` will cause funds to be locked in the adapter contracts

**Severity:** Low Risk

**Context:** L1OpUSDCBridgeAdapter.sol#L242-L244

**Description:** In the current implementation of USDC, transfers to `address(0)` are reverted. However, the `L1OpUSDCBridgeAdapter` and `L2OpUSDCBridgeAdapter` do not revert a cross-chain transfer to `address(0)` on the origin chain. Instead, the transfer only fails during the `mint` call on the receiving chain adapter. This discrepancy results in funds being permanently locked in the origin chain adapter contract, creating inconsistent behavior with the standard USDC implementation.

**Recommendation:** Consider reverting the transaction for cross chain transfers to `address(0)` in both `L1OpUSDCBridgeAdapter` and `L2OpUSDCBridgeAdapter`.

**Wonderland:** Fixed in commit eb625f95.

**Cantina Managed:** Fixed by implementing the auditor's recommendation.

### 3.2.4 L2 deploying message always fails and requires a manual replay to succeed

**Severity:** Low Risk

**Context:** L1OpUSDCFactory.sol#L67-L71, L2OpUSDCFactory.sol#L30-L61

**Description:** The contract `L1OpUSDCFactory` implements the function `deploy()`:

```
function deploy(
  address _l1Messenger,
  address _l1AdapterOwner,
  L2Deployments calldata _l2Deployments
) external returns (address _l1Adapter, address _l2Factory, address _l2Adapter) {
  // Checks that the first init tx selector is not equal to the `initialize()` function since  we manually
  // construct this function on the L2 factory contract
  if (bytes4(_l2Deployments.usdcInitTxs[0]) == _INITIALIZE_SELECTOR) revert IL1OpUSDCFactory_NoInitializeTx();

  // Update the salt counter so the L2 factory is deployed with a different salt to a different address and
↪  get it
  uint256 _currentNonce = ++deploymentsSaltCounter;

  // Precalculate the l1 adapter
  _l1Adapter = CrossChainDeployments.precalculateCreateAddress(address(this), _currentNonce);

  // Get the L1 USDC naming and decimals to ensure they are the same on the L2, guaranteeing the same standard
  IL2OpUSDCFactory.USDCInitializeData memory _usdcInitializeData =
    IL2OpUSDCFactory.USDCInitializeData(USDC_NAME, USDC_SYMBOL, USDC.currency(), USDC.decimals());

  // Use the nonce as salt to ensure always a different salt since the nonce is always increasing
  bytes32 _salt = bytes32(_currentNonce);
  // Get the L2 factory init code and precalculate its address
  bytes memory _l2FactoryCArgs = abi.encode(
    _l1Adapter,
    _l2Deployments.l2AdapterOwner,
    _l2Deployments.usdcImplementationInitCode,
    _usdcInitializeData,
    _l2Deployments.usdcInitTxs
  );

  // Send the L2 factory deployment tx
  _l2Factory = CrossChainDeployments.deployL2Factory(
    _l2FactoryCArgs, _salt, _l1Messenger, L2_CREATE2_DEPLOYER, _l2Deployments.minGasLimitDeploy
  );

  // Precalculate the L2 adapter address
  _l2Adapter = CrossChainDeployments.precalculateCreateAddress(_l2Factory, _L2_ADAPTER_DEPLOYMENT_NONCE);
  // Deploy the L1 adapter
  address(new L1OpUSDCBridgeAdapter(address(USDC), _l1Messenger, _l2Adapter, _l1AdapterOwner));

  emit L1AdapterDeployed(_l1Adapter);
}
```

This function deploys the L1 Adapter and initiates the deployment transactions for the L2 factory, L2 adapter, and L2 USDC through the L1 cross-chain messenger.

However, the L2 deployment message is failing every time. To resolve this state, the protocol team needs to manually call the `relayMessage()` function in the L2 CrossDomainMessenger contract with sufficient gas.

This issue is caused by Optimism Stack which due to the high gas requirements of the message is unable to execute the message which always ends up as a failed message in the L2.

The failed L2 deployment would not be visible to the L1 Adapter which would still allow users to use the `L1OpUSDCBridgeAdapter.sendMessage()` functions. However, these messages would fail in the L2 Messenger as the L2 adapter would not yet be deployed. Users would need to manually execute these messages as well once the L2 adapter is correctly deployed.

- **Relevant tests in Sepolia testnet:**

  The tests below show that messages that require more than `7.000.000` gas ends up failing and require to be replayed manually:

  - 1.000.000 gas OK: L1 TX, L2 TX
  - 4.000.000 gas OK: L1 TX, L2 TX
  - 6.000.000 gas OK: L1 TX, L2 TX
  - 7.000.000 gas FAILED: L1 TX, L2 TX
    - Failed relayed message: `0xe2d0eb696b72d068b3292b149dd715f5b9c172311889b123def3d6ed024525ac`.
  - 8.000.000 gas FAILED: L1 TX, L2 TX
    - Failed relayed message: `0x2459b02e9256f14036176fc53b8cea05c9a5c837b49a95ea643590d1816d4b8e`.

**Impact:** Low, as the user can simply replay the message in the L2 to deploy all the needed contracts.

**Likelihood:** High, as it will happen every time L2 contracts are deployed.

**Recommendation:** Consider enforcing a minimum `_l2Deployments.minGasLimitDeploy` amount in the `L1OpUSDCFactory.deploy()` function. Based in previous tests the amount of gas needed to deploy all the L2 contracts is 8.371.814. Consider setting this minimum amount to `9.000.000` gas.

On the other hand, ensure that the OP Stack works correctly with messages that require high amounts of gas to be executed.

It is also recommended to leave the L1 adapter in a "PRE-L2-DEPLOY" state after its deployment. This state would not allow to call `sendMessage()`. The L2 Adapter, in its constructor, could send a message to the L1 Adapter, notifying the L1 Adapter that the L2 was correctly deployed and setting the L1 adapter in a "WORKING" state that allows bridging. Notice though, that the messages sent from L2 to L1 can take an average of 7 days.

Finally, consider to split the deployment of the L2 contracts in 2 different messages. Each message should use less than `6.000.000` gas:

1. First message deploys the USDC implementation, USDC Proxy & the `FallbackProxyAdmin`.
2. Second messag deploys the `L2OpUSDCBridgeAdapter`.

**Wonderland:** Fixed in commit eb625f95.

**Cantina Managed:** Fix verified. The deployment message is sent through the Portal directly. Moreover, the gas costs of the L2 deployment were reduced to around `4.500.000` gas as now the USDC implementation is deployed manually in the L2.

### 3.2.5 Any call to `Adapter.sendMessage()` can be front-run by a malicious user submitting a 0 `_minGasLimit` parameter

**Severity:** Low Risk

**Context:** L1OpUSDCBridgeAdapter.sol#L218, L2OpUSDCBridgeAdapter.sol#L164

**Description:** The Adapter contracts implement the function `sendMessage()`:

```
/**
 * @notice Send signer tokens to other chain through the linked adapter
 * @param _signer The address of the user sending the message
 * @param _to The target address on the destination chain
 * @param _amount The amount of tokens to send
 * @param _signature The signature of the user
 * @param _deadline The deadline for the message to be executed
 * @param _minGasLimit Minimum gas limit that the message can be executed with
 */
function sendMessage(
  address _signer,
  address _to,
  uint256 _amount,
  bytes calldata _signature,
  uint256 _deadline,
  uint32 _minGasLimit
) external override {
  // Ensure messaging is enabled
  if (messengerStatus != Status.Active) revert IOpUSDCBridgeAdapter_MessagingDisabled();

  // Ensure the deadline has not passed
  if (block.timestamp > _deadline) revert IOpUSDCBridgeAdapter_MessageExpired();

  // Hash the message
  bytes32 _messageHash =
    keccak256(abi.encode(address(this), block.chainid, _to, _amount, _deadline, userNonce[_signer]++));

  _checkSignature(_signer, _messageHash, _signature);

  // Transfer the tokens to the contract
  IUSDC(USDC).safeTransferFrom(_signer, address(this), _amount);

  // Send the message to the linked adapter
  ICrossDomainMessenger(MESSENGER).sendMessage(
    LINKED_ADAPTER, abi.encodeWithSignature('receiveMessage(address,uint256)', _to, _amount), _minGasLimit
  );

  emit MessageSent(_signer, _to, _amount, MESSENGER, _minGasLimit);
}
```

As the `_minGasLimit` and the actual caller(`msg.sender`) is not included in the `_messageHash` any call to this function can be front-run by a malicious user submitting a 0 `_minGasLimit` parameter. Therefore, the transaction in the destination chain could fail with an out of gas error.

The USDC receiver would have to manually call the `relayMessage()` function in the CrossDomainMessenger contract with enough gas to be able to recover the funds in the destination chain.

**Impact:** Low, as there is no loss of funds.

**Likelihood:** Low, as this attack vector would amount to mere griefing without offering any incentive to the attacker.

**Recommendation:** Consider adding either `_minGasLimit` or `msg.sender` (the actual caller) to the computed `_messageHash`.

**Wonderland:** Fixed in commit eb625f95.

**Cantina Managed:** Fix verified. `_minGasLimit` was added to the computed `_messageHash`.

## 3.3 Informational

### 3.3.1 Prefer `abi.encodeCall` over less type-safe encoding methods

**Severity:** Informational

**Context:** USDCInitTxs.sol#L8-L24, L1OpUSDCBridgeAdapter.sol#L82-L86, L1OpUSDCBridgeAdapter.sol#L155-L157, L1OpUSDCBridgeAdapter.sol#L177-L179, L1OpUSDCBridgeAdapter.sol#L202-L204, L1OpUSDCBridgeAdapter.sol#L242-L244, L2OpUSDCBridgeAdapter.sol#L83-L85, L2OpUSDCBridgeAdapter.sol#L148-L150, L2OpUSDCBridgeAdapter.sol#L190-L192, CrossChainDeployments.sol#L35-L36

**Description:** The codebase uses `abi.encodeWithSignature` and `abi.encodeWithSelector` in multiple instances. These methods provide less type checking compared to `abi.encodeCall`, potentially leading to subtle errors that are hard to detect.

**Recommendation:**

1. Replace `abi.encodeWithSignature` with `abi.encodeCall` where possible.
2. If using `abi.encodeWithSignature` and an interface exists, switch to `abi.encodeCall`.
3. If no interface exists for `abi.encodeWithSignature`, create one and then use `abi.encodeCall`.
4. Replace all instances of `abi.encodeWithSelector` with `abi.encodeCall`.

Example:

```
// Instead of:
abi.encodeWithSignature('receiveMessage(address,uint256)', _to, _amount)

// Use:
abi.encodeCall(OpUSDCBridgeAdapter.receiveMessage, (_to, _amount))
```

This change enhances type safety, reduces the risk of errors, and improves code readability across all identified instances.

**Wonderland:** Fixed in commit eb625f95.

**Cantina Managed:** Fixed by implementing the auditor's recommendation.

### 3.3.2 Unused constant variable in `L1OpUSDCFactory`

**Severity:** Informational

**Context:** L1OpUSDCFactory.sol#L21

**Description:** The contract defines an unused *public* constant variable `L2_MESSENGER`.

**Recommendation:** Remove the unused `L2_MESSENGER` constant to improve clarity and reduce deployment gas costs.

**Wonderland:** Fixed in commit eb625f95.

**Cantina Managed:** Fixed in commit by implementing the auditor's recommendation.

### 3.3.3 Non-EIP-712 compliant message signing in the adapters

**Severity:** Informational

**Context:** L1OpUSDCBridgeAdapter.sol#L232-L236, L2OpUSDCBridgeAdapter.sol#L178-L182, OpUSDCBridgeAdapter.sol#L83-L87

**Description:** The USDC Bridge Adapters use a non-EIP-712 compliant method for message signing. The current implementation hashes the message data directly and then applies the Ethereum signed message hash, rather than using the structured data hashing defined in EIP-712.

In both L1 and L2 adapters, the message hash is created as follows:

```
bytes32 _messageHash =
  keccak256(abi.encode(address(this), block.chainid, _to, _amount, _deadline, userNonce[_signer]++));
```

This hash is then passed to the `_checkSignature` function:

```
function _checkSignature(address _signer, bytes32 _messageHash, bytes memory _signature) internal view {
  _messageHash = _messageHash.toEthSignedMessageHash();

  if (!_signer.isValidSignatureNow(_messageHash, _signature)) revert IOpUSDCBridgeAdapter_InvalidSignature();
}
```

The issue lies in the fact that this approach doesn't follow the EIP-712 standard for structured data hashing. As a result, when users are requested to sign a message, they will be presented with a seemingly random 32-byte value, rather than a human-readable structured message.

This lack of clarity could lead to user confusion and potential security risks, as users may be signing data without fully understanding its content or purpose. It might also make the signature-based bridging feature less usable or even unacceptable for security-conscious users.

**Recommendation:** To address this issue, it is strongly recommended to implement EIP-712 compliant structured data signing. This would involve:

1. Defining a structured data type for the message, for example:

```
// Self address and chainid are excluded because they are part of the domain separator.
struct BridgeMessage {
    address to;
    uint256 amount;
    uint256 deadline;
    uint256 nonce;
}
```

2. Implementing EIP-712 type hash:

```
bytes32 constant BRIDGE_MESSAGE_TYPEHASH = keccak256(
    "BridgeMessage(address to,uint256 amount,uint256 deadline,uint256 nonce)"
);
```

3. Introducing a hashing function to hash the EIP-712 structured data:

```
function hashMessageStruct(BridgeMessage memory message) internal pure returns (bytes32) {
    return keccak256(abi.encode(
        BRIDGE_MESSAGE_TYPEHASH,
        message.to,
        message.amount,
        message.deadline,
        message.nonce
    ));
}
```

4. Inheriting OpenZeppelin EIP-712 contract and using `_hashTypedDataV4()` internal function to get the final digest:

```
return _hashTypedDataV4(hashMessageStruct(message));
```

5. Passing the final output to `isValidSignatureNow()` function to check validity.

By implementing these changes, users will be presented with clear, structured data when signing messages, improving both security and usability of the bridge adapters.

**Wonderland:** Fixed in commit eb625f95.

**Cantina Managed:** Fixed per recommendation.

### 3.3.4 Dirty bits in precalculated `CREATE2` address

**Severity:** Informational

**Context:** CrossChainDeployments.sol#L47-L61

**Description:** The `precalculateCreate2Address` function in the CrossChainDeployments library uses inline assembly to efficiently calculate a contract address that would result from a `CREATE2` deployment. However, the function does not sanitize the resulting address, leaving "*dirty bits*" in the upper 96 bits of the 256-bit word representing the address.

The relevant code snippet is:

```
function precalculateCreate2Address(
  bytes32 _salt,
  bytes32 _initCodeHash,
  address _deployer
) internal pure returns (address _precalculatedAddress) {
  assembly ("memory-safe") {
    let _ptr := mload(0x40)
    mstore(add(_ptr, 0x40), _initCodeHash)
    mstore(add(_ptr, 0x20), _salt)
    mstore(_ptr, _deployer)
    let _start := add(_ptr, 0x0b)
    mstore8(_start, 0xff)
    _precalculatedAddress := keccak256(_start, 85)
  }
}
```

The `_precalculatedAddress` is assigned the raw result of `keccak256`, which occupies a full 256-bit word. The lower 160 bits represent the actual address, while the upper 96 bits may contain arbitrary data.

In normal Solidity usage, this isn't problematic because:

1. Solidity automatically masks addresses to 160 bits when they're used in address-typed variables or parameters.
2. ABI encoding (used for function calls and event emissions) also sanitizes addresses.

However, if this value were to be used directly in assembly code without proper masking, it could lead to unexpected behavior or vulnerabilities. Currently, there is no place in the codebase where the return value "_precalculatedAddress" is used in assembly, but any code changes or if this library is used somewhere else, it can be problematic.

**Proof of concept:** The following standalone contract demonstrates the issue:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract DirtyBitsPoC {
    function unsanitizedAddress() internal pure returns (address) {
        address result;
        assembly {
            // Set result to uint256.max (all bits set to 1)
            result := not(0)
        }
        return result;
    }

    function demonstrateIssue() public pure returns (uint256, uint256) {
        address addr = unsanitizedAddress();

        uint256 solidityValue;
        uint256 assemblyValue;

        // Use the address in normal Solidity context
        solidityValue = uint256(uint160(addr));

        // Use the address in assembly context
        assembly {
            assemblyValue := addr
        }

        return (solidityValue, assemblyValue);
    }
}
```

When deployed and `demonstrateIssue()` is called, it will return:

- `solidityValue`: `0xffffffffffffffffffffffffffffffffffffffff` (160 bits)
- `assemblyValue`: `0xffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff` (256 bits)

This demonstrates how the dirty bits are preserved when used in assembly but sanitized in Solidity contexts.

**Recommendation:** To mitigate this issue, consider implementing one of the following solutions:

1. Sanitize the address within the function:

```
_precalculatedAddress := and(keccak256(_start, 85), 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF)
```

2. Alternatively, use bit shifting to clear the upper bits:

```
_precalculatedAddress := shr(96, shl(96, keccak256(_start, 85)))
```

3. If performance is a critical concern and you choose not to sanitize the address, add a prominent comment to the function:

```
/// @dev WARNING: This function returns an unsanitized address with dirty bits in the upper 96 bits.
///      Do NOT use the return value directly in assembly without proper masking.
```

**Wonderland:** Fixed in commit eb625f95.

**Cantina Managed:** Fixed per the recommendation.

### 3.3.5 Multiple minters getting added to the L2 USDC contract must abide to bridge standards

**Severity:** Informational

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

**Description:** If a second minter is added to the L2 USDC implementation, these smart contracts provide no guarantee related to the correctness or the standard-abiding behaviour of the token.

Assuming a single minter, there is the assumption that the `total supply on L2 == total supply on L1` as there is always a 1-1 mapping between funds locked on L1 and funds locked on L2. This is loosely enforced in the smart contracts, where the `burnLockedAmount` is the minimum of the balance in the L1 adapter and the total supply in the L2 adapter.

However, If there are multiple minters, this adjusts the invariant to `total supply on L2 <= total supply on L1`, these smart contracts no longer provide guarantee that the entire burnAmount has actually been burnt.

If one of these additional minters does not build a bridge that adheres to the bridged USDC standard, the following property violation may be possible:

> Burn the amount of USDC held by the bridge that corresponds precisely to the circulating total supply of bridged USDC established by the supply lock.

Bridged USDC standard

**Recommendation:** Document the preference in the protocol for a single minter, and ensure that documentation related to the external bridge standard are explicit and available. Document the risks of not adhering to the standard as well.

**Wonderland:** Acknowledged the concerns related to multiple minters.

### 3.3.6 L2 to L1 withdrawal flow also requires a withdrawal proving and a withdrawal finalizing transaction sent in the L1 chain

**Severity:** Informational

**Context:** L2OpUSDCBridgeAdapter.sol#L83-L85, L2OpUSDCBridgeAdapter.sol#L148-150, L2OpUSDCBridgeAdapter.sol#L190-L192

**Description:** As per the Optimism Stack documentation withdrawals (transaction sent from L2 to L1) require the user to submit three transactions:

1. Withdrawal initiating transaction, which the user submits on L2. This is what is currently implemented in the `L2OpUSDCBridgeAdapter` contract through the different `sendMessage()` calls.
2. Withdrawal proving transaction, which the user submits on L1 to prove that the withdrawal is legitimate (based on a Merkle-Patricia trie root that commits to the state of the `L2ToL1MessagePasser`'s storage on L2).
3. Withdrawal finalizing transaction, which the user submits on L1 after the fault challenge period has passed, to actually run the transaction on L1.

The withdrawal proving and the withdrawal finalizing transactions are not implemented at smart contract level and require a backend that submits these transactions automatically in the L1 in order for the bridge to be fully operative.

**Recommendation:** In order for the bridge to be fully operative and allow communications from L2 to L1, ensure that there is a working backend that checks the initiated transactions in the `L2OpUSDCBridgeAdapter` contract and automatically submits the withdrawal proving and the withdrawal finalizing transactions in the L1.

**Wonderland:** Acknowledged.

**Cantina Managed:** Acknowledged.

### 3.3.7 USDC token name does not match Circle's USDC Bridged USDC Standard

**Severity:** Informational

**Context:** L1OpUSDCFactory.sol#L27

**Description:** The token name specified in the code does not match the recommended naming guidelines for the name as specified by the Bridged USDC Standard.

The name in the code is:

```
Bridged USDC
```

The name as mentioned in the Circle Documentation is:

```
Token Name: Bridged USDC (Third-Party Team)
```

Reference: Token Naming section of the Bridged USDC Standard.

**Recommendation:** Either add `Wonderland` to the token name or add a placeholder such that projects using this contract to make changes are aware that the token name should be edited.

**Wonderland:** Fixed in PR 138 by concatenating the USDC name and provided chain name on deploy in the `L1OPUSDCFactory.sol`.

**Cantina Managed:** Fixed by string concatenation to meet the above specified standard.

### 3.3.8 Incorrect natspec on the `stopMessaging` function

**Severity:** Informational

**Context:** L1OpUSDCBridgeAdapter.sol#L143, L1OpUSDCBridgeAdapter.sol#L93

**Description:** Incorrect natspec on the `stopMessaging` function suggests that the pausing of message passing is irreversible – when in fact it is the migration process / setting of the burn amount that is irreversible.

A counterpart function to the `stopMessaging` exists, named `resumeMessaging` that will allow messages to be transported through the bridge again.

**Recommendation:** Remove the `irreversible` natspec on the `stopMessaging` function and move it to the `setBurnAmount` function.

**Wonderland:** Fixed in PR 138 by removing irreversible messaging on the `stopMessaging` function.

**Cantina Managed:** Fixed with natspec removed on the `stopMessaging`. No natspec was added to the `setBurnAmount`.

### 3.3.9 The `L1OpUSDCFactory` contract is prone to human errors, potentially leaving the L2 USDC contract not completely initialized

**Severity:** Informational

**Context:** L1OpUSDCFactory.sol#L70, IL1OpUSDCFactory.sol#L20

**Description:** The `L1OpUSDCFactory.deploy()` function allows passing an array of USDC initialize transactions:

```
struct L2Deployments {
  address l2AdapterOwner;
  bytes usdcImplementationInitCode;
  bytes[] usdcInitTxs; // <----------------------
  uint32 minGasLimitDeploy;
}
```

The current USDC implementation contract have 4 different initialize functions:

- `initialize(string,string,string,uint8,address,address,address,address)`.
- `initializeV2(string)`.
- `initializeV2_1(address)`.
- `initializeV2_2(address[],string)`.

It is crutial that all the initialization functions are executed otherwise it could be exploited in multiple ways. The code below implemented in the `L2OpUSDCFactory` is prone to human errors (i.e. one of the initialization functions is left uninitialized):

```
// Execute the input init txs, use `_i+1` as revert argument since the first tx is already executed on the
↪  contract
for (uint256 _i; _i < _initTxs.length; _i++) {
  (bool _success,) = _usdc.call(_initTxs[_i]);
  if (!_success) {
    revert IL2OpUSDCFactory_InitializationFailed(_i + 1);
  }
}
```

**Recommendation:** Ensure that all the initialization functions are initialized. Inform the deployers that it is crutial to avoid leaving the USDC contract not fully initialized.

**Wonderland:** Acknowledged. The documentation was updated to inform any deployer of the steps that they must follow for a secure deployment.

**Cantina Managed:** Acknowledged.

### 3.3.10 Avoid using sequential nonce incrementation as it limits parallel signing

**Severity:** Informational

**Context:** L1OpUSDCBridgeAdapter.sol#L234, L2OpUSDCBridgeAdapter.sol#L180

**Description:** Currently, signature replay attacks are prevented in the `sendMessage()` implementation by incrementing the `userNonce` mapping:

```
// Hash the message
bytes32 _messageHash =
  keccak256(abi.encode(address(this), block.chainid, _to, _amount, _deadline, userNonce[_signer]++));

_checkSignature(_signer, _messageHash, _signature);
```

In order to give more flexibility to potential signers avoid increasing the `userNonce`. Instead, let the signer sign with any nonce and once that nonce is used, mark it as used so it can not be used again:

```
// User $\rightarrow$ Nonce $\rightarrow$ Used?
mapping(address $\rightarrow$ mapping(uint256 $\rightarrow$ bool)) public userNonces;

// Marking it as used
userNonces[_user][_nonce] = true;

// Checking it was not used before
require(!(userNonces[_user][_nonce]), "Nonce already used");
```

This approach gives more flexibility as:

1. There is no sequential dependency: In the original approach, the `userNonce` is incremented sequentially. This means that signers must keep track of the last used nonce and use the next one in sequence. If a nonce is missed or a transaction is delayed, it could create issues with subsequent transactions.
2. Allows for parallel signing: With the proposed approach, signers can use any nonce value, not just the next sequential one. This allows multiple transactions to be signed and processed in parallel without worrying about the exact order.

**Recommendation:** Consider implementing a more flexible nonce system by using a mapping to track used nonces per user. This will remove the sequential dependency and enable parallel signing of transactions.

**Wonderland:** Fixed in commit eb625f95.

**Cantina Managed:** Fix verified. The fix involved implementing the recommended solution of allowing signers to choose the nonce to be used. This nonce can only be used once. On the other hand, a `cancelSignature()` function (OpUSDCBridgeAdapter.sol#L124) was added that allows signers to mark any nonce as used. This function still needs to emit the nonce that is cancelled for transparency and to ease off-chain monitoring.

**Wonderland:** Fixed in commit 79f128ea.

**Cantina Managed:** Fix verified.

### 3.3.11 `L1OpUSDCFactory` does not guarantee that the deployed L2 USDC implementation is safe

**Severity:** Informational

**Context:** L1OpUSDCFactory.sol#L67-L71

**Description:** The `L1OpUSDCFactory.deploy()` function allows passing any `usdcImplementationInitCode` in its `L2Deployments calldata _l2Deployments` parameter. Consequently, any user can call this function and deploy a malicious L2 USDC implementation.

**Impact:** Low, as users are expected to validate that the deployed USDC contract is legit.

**Likelihood:** Low, as users are not expected to interact with any malicious implementation.

**Recommendation:** Introduce a centralized mechanism in the `L1OpUSDCFactory` to ensure only approved `usdcImplementationInitCode` are accepted. Maintain a mapping of valid `usdcImplementationInitCode` keccak hashes, which can only be updated by the contract owner.

Ensure that the keccak hash of the `usdcImplementationInitCode` passed in the `deploy()` function is whitelisted. If it is not, deny the deployment.

**Wonderland:** Acknowledged.

**Cantina Managed:** Acknowledged.

### 3.3.12 Missing burnAmount from MigrationComplete event

**Severity:** Informational

**Context:** L1OpUSDCBridgeAdapter.sol#L133

**Description:** Insufficient data to show a user how much money was burnt from calling the `burnLockedUSDC()` on the contract. The actual amount of USDC to be burnt is calculated as the minimum of the provided burn amount (L2 total supply) and the USDC balance of the L1Adapter contract, and can result in less than the L2 total supply.

This results in the following scenario:

1. Linked Adapter sets the `burnAmount` to 500 thus users assume that the amount of USDC that will be burnt is 500.
2. On calling `burnLockedUSDC`, the burnAmount is zeroed out, but the actual balance of USDC in the contract is only 200, so only 200 tokens are burnt.
3. Due to a lack of event emission proving this difference, it may not be obvious until later down the line that the initial amount submitted to burn was not reflective of the actual amount.

**Recommendation:** Add the `burnAmount` to the `MigrationComplete` event to ensure users are aware of how many tokens were burnt throughout this process.

**Wonderland:** Fixed in PR 138 by adding the burn amount in the `MigrationComplete` event.

**Cantina Managed:** Fixed. Fix converts the previously empty `MigrationComplete` event to emit with a `_burnAmount` argument.

### 3.3.13 Improvements on Fuzzing Suite

**Severity:** Informational

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

**Description:** The Wonderland OpUSDC codebase has a significant number of invariant tests. The below provides recommendations on improvements and enhancements that can be made to increase coverage.

- **Add relevant assertions to all empty `catch { }` clauses**

  Not checking anything within these clauses can result in missed corner cases. While it may be difficult to fully match all preconditions required for a successful call, start by documenting all the cases in which the attempted call my revert, such that implementing an assertion should not be too much effort.

  For example, take `id-14` – where the catch clause is empty:

```
    /// @custom:property-id 14
    /// @custom:property Incoming successful messages should only come from the linked adapter's
    function fuzz_l2LinkedAdapterIncommingMessages(uint8 _selectorIndex, uint256 _amount, address
↪  _address) public {
      _selectorIndex = _selectorIndex % 3;

      hevm.prank(l2Adapter.MESSENGER());
      if (_selectorIndex == 0) {
        try l2Adapter.receiveMessage(_address, _amount) {
          // Mint tokens to L1 adapter to keep the balance consistent
          hevm.prank(_usdcMinter);
          usdcMainnet.mint(address(l1Adapter), _amount);
          assert(mockMessenger.xDomainMessageSender() == address(l1Adapter));
        } catch {}
      [...]
```

  From here, if we add an `assert(false)` into the catch statement, we can quickly see there is a failure of `InvalidSender` because the sender is not correct. This showcases why adding assertions to the catch statement are important, and the failure to do so can provide a false sense of security.

```
  fuzz_l2LinkedAdapterIncommingMessages(uint8,uint256,address): failed!
    Call sequence:
      OpUsdcTest.fuzz_l2LinkedAdapterIncommingMessages(0,0,0x0)

  Traces:
  call 0x159932407dC2226A7EFAF9A9E7E224Be35537269::MESSENGER()
↪  (defi-wonderland-opUSD/test/invariants/fuzz/OpUSDC.t.sol:433)
   (MockBridge@0x4200000000000000000000000000000000000007)
  call HEVM::prank(0x0000000000000004200000000000000000000000000000000000000007)
↪  (defi-wonderland-opUSD/test/invariants/fuzz/OpUSDC.t.sol:433)
  call 0x159932407dC2226A7EFAF9A9E7E224Be35537269::receiveMessage(@0x00000000000000000000000000000000000
↪  0000, 0)
↪  (defi-wonderland-opUSD/test/invariants/fuzz/OpUSDC.t.sol:435)
  call MockBridge::xDomainMessageSender()() <no source map>
     (OpUsdcTest@0x00a329c0648769A73afAc7F9381E08FB43dBEA72)
  error Revert IOpUSDCBridgeAdapter_InvalidSender () <no source map>
```

- **Add specific invariant functions to test the transition of status**

  Id 18 tests that the status must either be one of the four message statuses which may work, however is a relatively basic check, considering the enum only has four possible states. Instead, consider adding specific invariant checks that only check the adjustment and changing of state.

- **Ensure invariants reflect realistic behaviour**

  The invariants may provide a false sense of security, depending on how they are worded. Some examples can be found below:

- `USDC proxy admin and token ownership rights **can only** be transferred during the migration to native flow (id-17)` – this is not, precisely, what the fuzzing suite is checking. The fuzzing suite rather, is checking that when the `transferUSDCRoles` function is called, that the owner and the admin addresses are updated, but the fuzzer provides no guarantees that these addresses can only change while calling this function.

- `Resume should be able to be set only by the owner and through the correct function (id-8)` – similar to above, this provides no guarantee that this is the **only** way that the owner address can change. This test merely checks that through calling the `resumeMessaging` function, the output / post-condition state is correct.
- `Incoming successful messages should only come from the linked adapter's (id-14)` – this is also, not exactly what the fuzzing test is currently testing. Regardless of `selectorIndex`, different `receiveX` functions are being called, and asserting that the `xDomainMessageSender` is equivalent to the `l1Adapter`. As the catch clauses are empty(related to bullet point 1), this invariant test does not actually test the failure case.
- `Set burn only if migrating (id-9)` – the test itself doesn't actually mention any form of migration or call any functions that are migration-related. The more accurate phrasing for this would be some variation of: `Set burnAmount on L1 if messengerStatus is UPGRADING`.
- Can receive USDC even if the state is not active (id-12)' – this property seemingly violates Circle's token standard expectations where it can "*pause USDC bridging to create a lock on total supply*".
- **Be careful using `hevm.prank` in the fuzzing suite**

  As highlighted in bullet point 1, issues can arise as only the subsequent call immediately after the prank call is to be pranked, which can result in calls looking like they were successful and actually failing. Care must be taken to ensure that all functions that are expected to run successfully have actually executed, and did not revert with error messages similar to "InvalidSender" or variations thereof.

  For example, the first hevm prank is not being applied to the `receiveMessage` call – it's applying to the call immediately after, the require statement, which doesn't need any form of

```
    /// @custom:property-id 14
    /// @custom:property Incoming successful messages should only come from the linked adapter's
    function fuzz_l1LinkedAdapterIncommingMessages(uint8 _selectorIndex, uint256 _amount, address
↪   _address) public {
      _selectorIndex = _selectorIndex % 2;
      if (_selectorIndex == 0) {
        hevm.prank(l1Adapter.MESSENGER());
        require(usdcMainnet.balanceOf(address(l1Adapter)) >= _amount);
        try l1Adapter.receiveMessage(_address, _amount) {
          // Mint tokens to L1 adapter to keep the balance consistent
          hevm.prank(_usdcMinter);
          usdcMainnet.mint(address(l1Adapter), _amount);
          assert(mockMessenger.xDomainMessageSender() == address(l2Adapter));
        } catch {assert(false);}
      }
      [...]
```

- **Use the coverage reports every time you run the fuzzer**

  Manual analysis of the coverage report is needed in order to ensure your invariants are correct, and your fuzzing suite is thoroughly investigating all branches you would expect it would.

  - **id-14 example**

```
405 |     |
406 |     |   /// @custom:property-id 14
407 |     |   /// @custom:property Incoming successful messages should only come from the linked adapter's
408 | *   |   function fuzz_l1LinkedAdapterIncommingMessages(uint8 _selectorIndex, uint256 _amount, address _address) public {
409 | *   |     _selectorIndex = _selectorIndex % 2;
410 | *   |     if (_selectorIndex == 0) {
411 | *   |     hevm.prank(l1Adapter.MESSENGER());
412 | *r  |       require(usdcMainnet.balanceOf(address(l1Adapter)) >= _amount);
413 | *   |       try l1Adapter.receiveMessage(_address, _amount) {
414 |     |         // Mint tokens to L1 adapter to keep the balance consistent
415 |     |         hevm.prank(_usdcMinter);
416 |     |         usdcMainnet.mint(address(l1Adapter), _amount);
417 |     |         assert(mockMessenger.xDomainMessageSender() == address(l2Adapter));
418 |     |       } catch {}
419 |     |     } else {
420 | *   |       try l1Adapter.setBurnAmount(usdcBridged.totalSupply()) {
421 |     |         // This will deprecate the adapter
422 |     |         _ghost_hasBeenDeprecatedBefore = true;
423 |     |         assert(mockMessenger.xDomainMessageSender() == address(l2Adapter));
424 |     |       } catch {}
425 |     |     }
426 |     |   }
```



  - **id-12 example**

```
366 |     |
367 |     |   /// @custom:property-id 12
368 |     |   /// @custom:property Can receive USDC even if the state is not active
369 | *   |   function fuzz_receiveMessageIfNotActiveL2(address _to, uint256 _amount) public agentOrDeployer {
370 |     |     // Precondition
371 | *r  |     require(_to != address(0) && _to != address(usdcMainnet) && _to != address(usdcBridged));
372 | *r  |     require(l2Adapter.isMessagingDisabled());
373 |     |
374 | *   |     _amount = clamp(_amount, 0, usdcBridged.balanceOf(_to) - 2 ** 255 - 1 - _amount);
375 |     |
376 |     |     // Set L1 Adapter as sender
377 |     |     mockMessenger.setDomainMessageSender(address(l1Adapter));
378 |     |
379 |     |     // cache balance
380 |     |     uint256 _toBalanceBefore = usdcBridged.balanceOf(_to);
381 |     |
382 |     |     hevm.prank(l2Adapter.MESSENGER());
383 |     |     // Action
384 |     |     try l2Adapter.receiveMessage(_to, _amount) {
385 |     |       // Postcondition
386 |     |       assert(usdcBridged.balanceOf(_to) == _toBalanceBefore + _amount);
387 |     |     } catch {
388 |     |       assert(usdcBridged.balanceOf(_to) == _toBalanceBefore);
389 |     |     }
390 |     |   }
```

Everything after the clamp suggests that the amount is clamped between unsafe values, such that Echidna cannot find any way to make these calls successful. Alternatively, a precondition to make these calls successful is missing.

- Use `try-catch` clauses and nest them if needed

  In id-11, the assumption is that the first call to `migrateToNative` is successful, and therefore the fuzzer should try to call the function a second time with the same arguments.

```
    /// @custom:property-id 11
    /// @custom:property Upgrading state only via migrate to native, should be callable multiple times
↪   (msg fails)
    function fuzz_migrateToNativeMultipleCall(address _burnCaller, address _roleCaller) public {
      // Precondition
      // Insure we haven't started the migration or we only initiated/is pending in the bridge
      require(
        l1Adapter.messengerStatus() == IL1OpUSDCBridgeAdapter.Status.Active
          || l1Adapter.messengerStatus() == IL1OpUSDCBridgeAdapter.Status.Upgrading
      );

      require(_burnCaller != address(0) && _roleCaller != address(0));

      // As the bridge would relay and execute the migration atomically, including deprecating l1adapter,
↪   we need to prevent
      // it from relaying the message to test this property
      mockMessenger.pauseMessaging();

      // Action
      // 11
      try l1Adapter.migrateToNative(_burnCaller, _roleCaller, 0, 0) {
        assert(l1Adapter.messengerStatus() == IL1OpUSDCBridgeAdapter.Status.Upgrading);
      } catch {}

      // try calling a second time
      try l1Adapter.migrateToNative(_burnCaller, _roleCaller, 0, 0) {}
      catch {
        assert(false);
      }
    }
```

  This poses a few problems:

  1. With no checks on the `catch` clause on the first call, it is unclear whether the first attempted migrateToNative was even successful.
  2. Assuming that the first `migrateToNative` had failed, there is no point to try to call the same function with the same arguments immediately afterward.
  3. The empty try statement on the second migrateToNative only checks the call is successful, and not whether the state had actually changed

- **Be decisive of what functions are public and private**

  Echidna by default will run any public or external function as part of its fuzzing suite. For this particular codebase, this is why functions under the "Expose target contract selectors" are being fuzzed as well. As there are no assertions in either of these functions, this would suggest the sole purpose is to generate valid inputs and/or valid helpers, in which case these functions should be *either* marked

internal/private or added to the Echidna/Medusa filter list to not fuzz these directly.

This is the reason that you see the following assertion while running the test suite, which will not actually help generate more state(s) because there are no explicit assertions here:

```
generateCallAdapterL1(uint256,address,address,uint256,uint256,uint32,uint32): passing
```

**Recommendation:** The files mentioned below contains recommendations for additional invariants that can be tested.

- `test/integration/Factories.t.sol.test_deployAllContracts`
- `test/invariants/fuzz/OpUSDC.t.sol.fuzz_testDeployments`

Consider adding the following checks, according to the Circle deployment checks:

- `implementation` is the address of the implementation contract.
- `version` is 2.
- `totalSupply` of token immediately after deployment is zero.
- `initialized` is `true`.

**Wonderland:** Wonderland partially fixed some of the concerns mentioned above, and will continue to make changes in future iterations.

**Cantina Managed:** Partially fixed in PR 136, and will continue to test as codebase grows.