OpenZeppelin | security

The Compact Audit

Uniswap

June 30, 2025

Table of Contents

Summary

Type	DeFi
Timeline	From 2025-04-28 To 2025-06-03
Languages	Solidity

Total Issues	31 (25 resolved, 2 partially resolved)
Critical Severity Issues	0 (0 resolved)
High Severity Issues	2 (2 resolved)
Medium Severity Issues	3 (3 resolved)
Low Severity Issues	7 (6 resolved)
Notes & Additional Information	19 (14 resolved, 2 partially resolved)

Scope - Phase 1

OpenZeppelin audited the Uniswap/the-compact repository at commit 102fa06.

In scope were the following files:

```
src
├── TheCompact.sol
├── interfaces
│   ├── IAllocator.sol
│   ├── IEmissary.sol
│   ├── ITheCompact.sol
│   └── ITheCompactClaims.sol
├── lib
│   ├── AllocatorLib.sol
│   ├── AllocatorLogic.sol
│   ├── BenchmarkERC20.sol
│   ├── ClaimHashFunctionCastLib.sol
│   ├── ClaimHashLib.sol
│   ├── ClaimProcessor.sol
│   ├── ClaimProcessorFunctionCastLib.sol
│   ├── ClaimProcessorLib.sol
│   ├── ClaimProcessorLogic.sol
│   ├── ComponentLib.sol
│   ├── ConstructorLogic.sol
│   ├── ConsumerLib.sol
│   ├── DepositLogic.sol
│   ├── DepositViaPermit2Lib.sol
│   ├── DepositViaPermit2Logic.sol
│   ├── DirectDepositLogic.sol
│   ├── DomainLib.sol
│   ├── EfficiencyLib.sol
│   ├── EmissaryLib.sol
│   ├── EmissaryLogic.sol
│   ├── EventLib.sol
│   ├── Extsload.sol
│   ├── HashLib.sol
│   ├── IdLib.sol
│   ├── MetadataLib.sol
│   ├── MetadataRenderer.sol
│   ├── RegistrationLib.sol
│   ├── RegistrationLogic.sol
│   ├── TheCompactLogic.sol
│   ├── TransferBenchmarkLib.sol
│   ├── TransferBenchmarkLogic.sol
│   ├── TransferFunctionCastLib.sol
│   ├── TransferLib.sol
│   ├── TransferLogic.sol
│   ├── ValidityLib.sol
```

```
│   └── WithdrawalLogic.sol
└── types
    ├── BatchClaims.sol
    ├── BatchMultichainClaims.sol
    ├── Claims.sol
    ├── CompactCategory.sol
    ├── Components.sol
    ├── DepositDetails.sol
    ├── EIP712Types.sol
    ├── EmissaryStatus.sol
    ├── ForcedWithdrawalStatus.sol
    ├── Lock.sol
    ├── MultichainClaims.sol
    ├── ResetPeriod.sol
    └── Scope.sol
```

Scope - Phase 2

In the second phase, we performed a diff audit between BASE commit 102fa06 and HEAD commit aba2e2f. This audit was focused on new features and code refactors.

System Overview

The Compact is an ownerless, on-chain protocol built around the ERC-6909 token standard that enables users to deposit ERC-20 or native tokens into *resource locks*, which are represented by a unique ERC-6909 token ID. These tokens allow their holders (called *sponsors*) to create *compacts*, which are signed commitments permitting an *arbiter* to execute payouts to *claimants* once the specified criteria are met. The protocol is chain-agnostic and supports both single-chain and multichain flows for single and batch claims.

Resource Locks: When a depositor locks tokens, a fungible ERC-6909 token is minted, encoding four parameters in its ID: the underlying token address, an *allocator ID*, a *scope* (single-chain or multichain), and a *reset period*. Depositors assign an *allocator* at deposit time, who must co-sign any future transfers, withdrawals, or compacts. If the allocator becomes unresponsive, the depositor may trigger a *forced withdrawal* after the reset period elapses.

Compacts and Claims: A sponsor holding ERC-6909 tokens can create a compact by either signing an EIP-712 payload or registering the claim hash on-chain. Each compact designates an arbiter responsible for verifying the compact's conditions and submitting a claim payload. Claim payloads include the sponsor's signature (if not registered), allocator data, nonce, expiration, lock ID(s), allocated amount(s), and an array of *Components* that specify how to distribute tokens—either as ERC-6909 transfers, lock conversions, or withdrawals of underlying assets.

Allocators: Every resource lock is governed by a registered *allocator*, whose duties are to:

- prevent double spending by ensuring that sponsors cannot allocate the same locked balance to multiple compacts
- validate standard ERC-6909 transfers (via `IAllocator.attest`)
- authorize claim requests on behalf of claimants (via `IAllocator.authorizeClaim`)
- manage nonces to prevent replays

Arbiters: Arbiters verify that the conditions of a compact have been met (implicitly or via supplied witness data) and then call the appropriate `claim` function. They construct claim payloads indicating which claimants receive which amounts and by which method (transfer, conversion, or withdrawal).

Fillers: Fillers supply the necessary resources or collateral to satisfy the compact's conditions.

Emissaries: Sponsors may optionally assign an *emissary* per `lockTag`, who can serve as a fallback signer if the sponsor cannot produce an ECDSA or EIP-1271 signature.

Security Model and Trust Assumptions

The Compact's security especially hinges on the correct behavior of allocators and arbiters, but also on that of emissaries and relayers. Below are the primary assumptions that ensure funds remain safe (no theft or under-collateralization) and live (valid claims eventually pay out).

Allocators

- Allocators must ensure that any compact submitted by an arbiter is fully backed by locked assets.
- Allocators should not arbitrarily block valid actions. If they do censor a legitimate compact or withdrawal, sponsors can make a forced withdrawal after the reset period.

Arbiters

- Sponsors and Claimants trust the arbiters to only process legitimate claims where conditions were met.
- Claimants depend on arbiters to act without undue delay to not conflict with the reset period.
- The claim payload's component list (which encodes target addresses and amounts) must be constructed correctly. Malicious modifications could misallocate funds.
- Arbiters are expected to be aware of upcoming chain forks before attempting to process a claim. A change in chain ID due to a chain fork will result in claim verification failure.

Fillers

- Fillers are expected to make educated decisions on whether a claim's condition is safe to fulfill, especially considering the reset period of the resource lock.
- When a sponsor assigns an emissary or uses EIP-1271 signature verification, fillers should assess whether these mechanisms provide sufficient equivocation resistance.
- Fillers should also be aware that sponsors can unilaterally change their account's code using EIP-7702, which may affect the behavior of claim authorization.

- Additionally, sponsors may assign or update emissaries at any time, either immediately (if none is currently set) or after a scheduled delay. These changes can influence the validity of the arbiter's `claim` execution and may cause claimants to miss out on receiving the compact's agreed token and amount.

Emissaries

- Emissaries, as assigned by sponsors, must only vouch for legitimate compacts. If compromised, they could authorize fraudulent claims.
- To guard against rapid malicious swaps, emissaries cannot be reassigned until after the resource lock's reset period elapses.

Relayers

Relayers are not a specific role within the Compact code. However, they can be used as part of gasless Permit2 deposits, to register claims, or to invoke claims through the arbiter. They are assumed to forward messages without alteration.

Additional Considerations

- All parties involved are assumed to follow best practices regarding account safety.
- The Compact expects the Permit2 contract to be deployed at a hard-coded address.
- The Compact expects to be deployed at the same address across chains.
- The allocator registration proof depends on the chain's `create2` address hashing.
- Tokens with fees on transfer might be cheaper to manage by transferring the ERC-6909 token.
- Rebasing tokens should be deposited through their wrapped, non-rebasing version.

For the complete documentation, please refer to the following README.

High Severity

H-01 Single Emissary Configuration Allows Users to Set Emissaries for Others

The Emissary role serves as a secondary verification mechanism that allows a designated individual (the emissary) to authorize claims on behalf of another user (the sponsor). This role is critical for ensuring trust and accountability within the system, as each user should independently control their emissary assignments.

Currently, due to incorrect memory management in the `_getEmissaryConfig` function, the storage slot intended to uniquely identify each user's emissary configuration is computed improperly. Specifically, during slot calculation, the `lockTag` overwrites the sponsor's address in memory, causing the final computed slot to ignore the sponsor and rely solely on `_EMISSARY_SCOPE` and `lockTag`. Consequently, all users sharing the same `lockTag` unintentionally share the same emissary configuration. An attacker exploiting this can set a malicious emissary who approves all claims, bypassing proper sponsor authorization and potentially granting unauthorized control over claims processing.

Consider correcting the memory overwrite in the `_getEmissaryConfig` function by correctly including the sponsor's address in the storage slot computation. This will ensure that each emissary configuration is uniquely identifiable.

Update: *Resolved in pull request #113 at commit 3caea0b.*

H-02 Bypassing Element Verification Allows for Unauthorized Manipulation of Exogenous Claim Data

The protocol utilizes cryptographic hashes to validate claim requests. Typically, each element of an exogenous multichain claim comprises an arbiter, token identifiers, allocated amounts, and other crucial parameters. These are collectively hashed and signed by a sponsor, ensuring that only authorized arbiters with legitimate claims can perform token transfers. In the context of exogenous multichain claims, a claim is composed of a main element (the "current element") and possibly multiple additional chain elements. All these elements' hashes must match

exactly what the sponsor had originally signed, preventing unauthorized alterations. Furthermore, a "witness" field exists, meant primarily for additional claim data, but its content is not strictly validated, enabling it to be repurposed under certain conditions.

The issue arises because the current element's hash, which explicitly includes critical details such as arbiter address, token identifiers, and allocated amounts, may not be incorporated into the final hash computation if the provided `chainIndex` is out of bounds. Specifically, the `toExogenousMultichainClaimMessageHash` function skips inserting the current element's hash into memory if the `chainIndex` does not align with the `additionalChains` array boundaries. Normally, this omission would result in a mismatch between the final computed hash and the sponsor-signed hash, since one more word is included during the hash calculation.

However, an attacker can exploit the `witness` field—which is stored at a predictable memory offset—to manually insert the last `additionalChain` element's hash, thereby reconstructing precisely the original signed hash. By doing this, the attacker effectively substitutes the legitimate current element (containing correct arbiter and amounts) with maliciously crafted data, including their own address as an arbiter address and arbitrary token IDs and allocation amounts, since the real data is not inserted into the claim hash. While allocator approval is technically required to execute the claim, an attacker can monitor legitimate transactions performed by the genuine arbiter and reuse valid allocator authorization data in their own front-run transaction. Since the allocator's logic is considered a black box and likely does not reconstruct the full claim hash for validation, it might inadvertently permit this unauthorized transaction.

Note that this scenario only works with a total of five elements because the witness argument is placed in the fifth word relative to the `m` memory pointer.

Consider explicitly validating that the current element's hash is always included in the final hash computation by ensuring that `extraOffset` is non-zero when expected. In addition, the protocol should prohibit attackers from manipulating memory offsets and prevent the `witness` field from being misused to inject maliciously constructed claim elements into the hash computation.

Update: *Resolved in pull request #113 at commit 08f2471.*

Medium Severity

M-01 Activation and Compact Typehashes Differ from Typestring Used in `writeWitnessAndGetTypehashes`

In the `writeWitnessAndGetTypehashes` function of the `DepositViaPermit2Lib` library, when the caller **does not** supply a witness (e.g., an allocated transfer), the library appends ")TokenPermissions(address token," one byte too early, leaving the in-memory typestring ending "Mandate)TokenPermissions..." instead of the syntactically valid "Mandate()", defined in the EIP-712. The constant pre-images (#1 and #2) used to derive `activationTypehash` and `compactTypehash` also omit any mandate reference.

As a result, the typehash values returned by `writeWitnessAndGetTypehashes` are computed from data that does **not** match the typestring subsequently passed to `Permit2.permitWitnessTransferFrom`, while the `toMessageHashWithWitness` function that is later used during the claim reconstructs the compact typehash from the *correct* "...Mandate()...." string. This discrepancy means that the hashes used during `depositERC20AndRegisterViaPermit2` execution and those used during later claim/validation are derived from different byte sequences.

Consider unifying the constant pre-images used when the witness is empty with the exact bytes written to memory, including an empty "Mandate()", so that activation and compact typehashes are derived from identical data throughout the protocol. In addition, consider correcting "Mandate)" to "Mandate()" in `writeWitnessAndGetTypehashes` and adding tests that compare the computed typehashes against those rebuilt in `toMessageHashWithWitness`.

Update: *Resolved in pull request #113 at commit c8b7c64, and in commits 41a2138 and 591f829. The Uniswap team stated:*

> *We determined that even if `Compact(address sponsor,...,Mandate mandate)Mandate()` (where `Mandate` has zero members) would be valid, it's better to simply omit it altogether for both readability and compatibility with existing tooling (support seems inconsistent). As such, we've refactored this area of code to only include the `,Mandate mandate)Mandate(` (and accompanying closing paren) only if the provided witness typestring length is nonzero.*

M-02 Possible Replay of `register*For` Calls Allows Bypassing Signature Expiration in `hasValidSponsorOrRegistration`

The `register*For` functions record the timestamp when a claim is authorized by a sponsor, which determines the validity period of that authorization. According to the documentation, a claim becomes inactive once the associated duration expires, requiring the sponsor to provide a new signature to execute the claim.

However, currently, the `register*For` functions do not prevent the replaying of past registrations, allowing arbitrary users to refresh the timestamp and extend the sponsor's authorization artificially. As a result, the sponsor's authorization step can be bypassed by replaying old registrations, effectively reducing the intended two-step verification (sponsor and allocator) to a single-step process. This allows a potentially malicious allocator, possibly collaborating with an arbiter, to execute actions without valid authorization from the sponsor, for actions that are considered expired.

Consider implementing protections within the `register*For` functions to prevent replaying or reusing sponsor signatures, thereby ensuring adherence to the documented rules for claim expiration.

Update: *Resolved at commit b7eef90. The Uniswap team stated:*

> *We have refactored this functionality to remove the notion of a registration duration.*

M-03 Target Address in `setNativeTokenBenchmark` Is Always the Zero Address

The `setNativeTokenBenchmark` function should derive a 20-byte recipient by hashing a caller-supplied salt with the contract address and shifting the 32-byte result right by 96 bits. The shift removes the upper 12 bytes, leaving an address that is intended to receive two benchmark transfers. The address zero is sometimes used as a burn address on Ethereum, which means that its balance is non-zero.

The implementation reverses the operands to the `shr` opcode: it shifts the literal `96` by the 256-bit hash instead of shifting the hash by 96. Since the shift distance far exceeds 96 for almost all hashes, the result is 0, and the truncated 20-byte slice becomes the zero address. A pre-check performed to ensure that this address's balance is zero then causes the benchmark

to revert in every call. As the benchmark value is not initialized, the `ensureBenchmarkExceeded` function will attempt to compare zero to `gas()`, causing the check to always pass.

Consider swapping the operands of the `shr` opcode so that the 256-bit hash is shifted right by 96 bits, producing a valid 20-byte address and enabling the benchmark transfers to execute as intended.

Update: *Resolved in pull request #113 at commit 90f0fce.*

Low Severity

L-01 Event Signature Mismatch During Claim

The signature calculation for the `Claim` event omits the `uint256` type of the `nonce` parameter, which is logged in the `emitClaim` function. As a result, the generated event signature does not match the actual event being emitted, which will break the off-chain decoding of the `Claim` event by tools that rely on accurate signature matching.

Consider correcting the event signature.

Update: *Resolved in pull request #65 at commit 1b01b6a.*

L-02 JSON Injection Enables Spoofing of Tokens

The `toURI` function of the `MetadataLib` library allows users to query token information by `Lock` and ID through the `uri` function in a JSON response format. When building the JSON response, the token information, such as the symbol and name are fetched and concatenated in a JSON syntax.

However, there is no escaping performed on the received token data [1, 2]. This enables a malicious token to inject misleading data into the response, possibly deceiving a user. For instance, the `name()` response of a token could escape the double quote context and append another `trait_type` and `value` that would lead to a second "Token Address" attribute with a different value.

Consider using Solady `LibString` or OpenZeppelin Contracts Strings.sol to escape the string inputs to prevent JSON injection.

Update: *Resolved in pull request #65 at commit 87ed6fa.*

L-03 No Fallback Recipient in Forced Withdrawal

After a user has enabled a forced withdrawal, they can call `forcedWithdrawal` to send the locked funds to a `recipient`. While during deposits [1, 2, 3] this `recipient` address is replaced with `msg.sender` in case the address is zero, this fallback is not performed during the withdrawal.

Consider applying the same `usingCallerIfNull` function to the `recipient` during withdrawal to prevent the accidental loss of funds.

Update: *Resolved in pull request #113 at commit 1476fac.*

L-04 Native Token Deposit Not Bounded in `batchDeposit*ViaPermit2`

The `batchDepositViaPermit2` and `batchDepositAndRegisterViaPermit2` functions accept a sponsor-signed Permit2 authorization and rely on a third-party executor—typically a relayer or service—to broadcast the transaction that performs multiple ERC-20 deposits and, optionally, attaches a native-currency deposit.

However, since the native amount is not included in the signed data, an observer can copy the relayer's pending transaction, lower `msg.value` to 1 wei (or omit it), and broadcast the clone first. The call still passes signature checks, consumes the permit nonce, and causes the legitimate transaction to revert because its nonce is spent. The expected native funds then remain with the relayer, who must send a separate transaction to refund the sponsor or complete the deposit, incurring extra gas and operational overhead.

Consider binding the native token and its exact amount in the sponsor-signed data so that any mismatch triggers an immediate revert before the nonce is consumed.

Update: *Acknowledged, not resolved. The Uniswap team stated:*

> *The `batchDepositAndRegisterViaPermit2` function allows an activator address to be specified, preventing third parties from using the permit data for frontrunning. With respect to the `batchDepositViaPermit2` function, we recognize the inconvenience but do not plan to address it at this time.*

L-05 Incorrect Gas Benchmarking for Native Token Transfers

The Ethereum Virtual Machine (EVM) charges different gas fees when an address is accessed for the first time in a transaction ("cold") versus any subsequent access in the same transaction ("warm"), as defined in EIP-2929. A cold access costs an additional 2,600 gas, while a warm access incurs only 100 gas. If the destination account does not yet exist, the first transfer also triggers a one-off 25,000 gas account-creation charge, as defined in EIP-161. A proper benchmark should therefore measure a cold transfer followed by a warm transfer to capture the expected 2,500 gas spread.

The `setNativeTokenBenchmark` function's current benchmarking routine invokes `BALANCE(target)` before performing two transfers of 1 wei to the same address in a single transaction. Since `BALANCE` warms the address, both of the subsequent transfers are treated as warm, eliminating the intended 2,500 gas differential. Consequently, when the target account already exists (i.e., nonce or code are non-zero), the recorded result is 2,500 gas lower than the true cold-access cost, causing the internal assertion that the two costs must differ to fail and the benchmark to revert. When the account is created during the call, the cold surcharge is still omitted, again understating the cost by 2,500 gas (but the benchmark will not revert, due to the additional 25,000 charge for the first transfer).

Consider removing or moving the `BALANCE` call so that the first transfer is truly cold. In addition, to ensure that users provision sufficient gas for all cases, consider calculating the benchmark against the most expensive path—account creation plus cold access.

Update: *Resolved at commit 90f0fce and commit c6b161c.*

L-06 Underflow via Token Hook Manipulation During Batch Claims

During batch claims, the protocol determines how many tokens a user has withdrawn by comparing the contract's own token balance before and after the `transfer` function is called. The difference is used to calculate the amount transferred to the receiver. This method assumes that the contract's balance will decrease during the withdrawal.

However, if the token implements hooks—such as those in ERC-777 or similar standards—a malicious receiver can exploit this behavior. For example, after receiving tokens from the `TheCompact` contract, the receiver can use a token hook to send back the same amount plus one token to the contract during the execution of `withdraw`. This results in the contract's

post-withdrawal balance appearing greater than pre-withdrawal balance, causing the subtraction to underflow and the transaction to revert. This results in other recipients failing to receive their tokens.

Consider adding documentation or including an underflow check and then proceeding with the release to avoid unexpected reverts.

Update: *Resolved in pull request #113 at commit 7d287bf.*

L-07 Inconsistent No-Witness Registration Handling - Phase 2

The `registerFor` function and similar registration functions compute the claim hash through `deriveClaimHashAndRegisterCompact`. Currently, this function always includes the witness in the hash calculation, even when the `typehash` is `COMPACT_TYPEHASH`, which indicates that no witness should be included. Conversely, `depositNativeAndRegisterFor` and related functions utilize `toClaimHashFromDeposit`, which correctly omits the witness in no-witness scenarios.

Consider modifying `deriveClaimHashAndRegisterCompact` to exclude the witness from hash computation in no-witness scenarios.

Update: *Resolved in pull request #143 at commit 510a434.*

Notes & Additional Information

N-01 Potential Bit Overlap in `toLockTag`

The `toLockTag` function generates a `bytes12` locktag value from an allocator ID, scope, and reset period. While the allocator ID is a `uint96` value, it is expected to only use the lower 92 bits, leaving the upper 4 bits for the scope and reset period. However, the upper 4 bits of the allocator ID are not cleared. Currently, this is not a problem since this function is only used with an ID that comes from the `usingAllocatorId` function.

To protect against bit manipulation in potential future uses in different contexts, consider clearing the upper bits of the allocator ID.

Update: *Resolved in pull request #123 at commit fb7605b.*

N-02 Unused Code

Throughout the codebase, multiple instances of unused code were identified:

Imports

- `IAllocator` in `AllocatorLib.sol`
- `ResetPeriod` in `ClaimHashLib.sol`
- `Scope` in `ClaimHashLib.sol`
- `ValidityLib` in `ClaimProcessorLogic.sol`
- `TransferComponent` in `ComponentLib.sol`
- `Lock` in `ConstructorLogic.sol`
- `ResetPeriod` in `ConstructorLogic.sol`
- `Scope` in `ConstructorLogic.sol`
- `Scope` in `DepositViaPermit2Logic.sol`
- `ResetPeriod` in `DirectDepositLogic.sol`
- `Scope` in `DirectDepositLogic.sol`
- `Scope` in `EmissaryLib.sol`
- `IEmissary` in `EmissaryLib.sol`
- `IAllocator` in `EmissaryLogic.sol`
- `ResetPeriod` in `EmissaryLogic.sol`
- `Scope` in `EmissaryLogic.sol`
- `ResetPeriod` in `EmissaryStatus.sol`
- `TransferComponent` in `HashLib.sol`
- `TransferFunctionCastLib` in `HashLib.sol`
- `CompactCategory` in `IdLib.sol`
- `ResetPeriod` in `RegistrationLogic.sol`
- `Lock` in `TheCompact.sol`
- `ConstructorLogic` in `TransferBenchmarkLib.sol`
- `IdLib` in `TransferBenchmarkLib.sol`
- `BenchmarkERC20` in `TransferBenchmarkLib.sol`
- `TransferComponent, ComponentsById` in `TransferFunctionCastLib.sol`
- `ConstructorLogic` in `TransferLib.sol`
- `TransferComponent` in `TransferLogic.sol`

`using` Statements:

- `using ValidityLib for bytes32` in `TransferLogic.sol`
- `using IdLib for ResetPeriod` in `ValidityLib.sol`
- `using EfficiencyLib for uint256` in `ValidityLib.sol`
- `using EfficiencyLib for ResetPeriod` in `ValidityLib.sol`
- `using ValidityLib for uint256` in `ValidityLib.sol`
- `using FixedPointMathLib for uint256` in `ValidityLib.sol`
- `using ClaimProcessorLib for uint256` in `ClaimProcessorLogic.sol`
- `using ClaimProcessorFunctionCastLib for function`s with 6 arguments in `ClaimProcessorLogic.sol`
- `using HashLib for uint256` in `ClaimProcessorLogic.sol`

Functions

- `toBatchMessageHash` in `HashLib.sol`
- `toAllocatorIdIfRegistered` in `IdLib.sol`
- `toCompactFlag` in `IdLib.sol`
- `toId` in `IdLib.sol`
- `readDecimalsWithDefaultValue` in `MetadataLib.sol`

Consider removing any code that is no longer used to allow for a leaner code footprint and ease of maintenance.

Update: *Resolved in pull request #117 at commit f236911.*

N-03 Lack of Security Contact

Embedding a dedicated security contact (email or ENS) in a smart contract streamlines vulnerability reporting by letting developers define the disclosure channel and avoid miscommunication. It also ensures third-party library maintainers can quickly reach the right person for fixes and guidance.

Consider adding a NatSpec comment containing a security contact above each contract, library, and interface definition. Using the `@custom:security-contact` convention is recommended as it has been adopted by the OpenZeppelin Wizard and ethereum-lists.

Update: *Resolved in pull request #127 at commit 13f2e3a.*

N-04 Magic Numbers

The codebase makes heavy use of `calldata` and `memory` pointers and offsets that are propagated to other functions. However, these are not explicitly documented such that it is more difficult to reason about the code. For instance:

- In `ClaimProcessorLogic.sol`
  - line 53
  - line 67
  - line 82
  - line 100
- In `DepositViaPermit2Logic.sol`
  - line 70
  - line 81
  - line 86
  - line 137
  - line 146
  - line 156
  - line 203
  - line 219
  - line 223
  - line 300
  - line 322
- In `RegistrationLogic.sol`
  - line 66
  - line 80
  - line 100
- In `HashLib.sol`
  - line 537-538 adds an offset to the additional offset provided [1, 2]. One offset addition can likely be simplified.

To improve the clarity and readability of the codebase, consider documenting which struct is being referred to by which pointer value, and whether any other offsets, lengths, or signature bytes are being accounted for in the value.

Update: *Acknowledged, not resolved.*

N-05 Custom Errors in `require` Statements

Since Solidity version `0.8.26`, custom error support has been added to `require` statements. While initially, this feature was only available through the IR pipeline, Solidity `0.8.27` has extended support to the legacy pipeline as well.

Throughout the codebase, multiple instances of `if-revert` statements that could be replaced with `require` statements were identified:

- lines 55-57 in `BenchmarkERC20.sol`
- lines 221-223 in `ComponentLib.sol`
- lines 328-330 in `ComponentLib.sol`
- lines 146-148 in `EmissaryLib.sol`
- lines 168-170 in `EmissaryLib.sol`
- lines 187-189 in `EmissaryLib.sol`

For conciseness and gas savings, consider replacing `if-revert` statements with `require` statemetns.

Update: *Acknowledged, not resolved.*

N-06 Use Custom Errors

Since Solidity version 0.8.4, custom errors provide a cleaner and more cost-efficient way to explain to users why an operation failed.

Throughout the codebase, multiple instances of string-based revert statements were identified:

- The `revert("Unknown reset period")` statement
- The `revert("Unknown scope")` statement

For conciseness and gas savings, consider replacing string-based revert messages with custom errors.

Update: *Resolved at commit 5eb33da. The revert statements have been removed.*

N-07 Function Visibility Not Always Properly Defined

The codebase utilizes many libraries and inherited contract components that interact with each other. A function's visibility is essential for understanding the scope of its use. However, a few functions are set to `internal` even though they are only used within the same library or contract:

- `_validateSponsor`
- `_validateAllocator` - 1
- `_validateAllocator` - 2
- `_buildIdsAndAmounts`
- `verifyAndProcessComponents`
- `toBatchTransferMessageHashUsingIdsAndAmountsHash`
- `toCompactFlag`
- `toString` - 1
- `toString` - 2
- `toAttributeString`
- `isValidECDSASignatureCalldata`
- `isValidERC1271SignatureNowCalldataHalfGas`

Consider correcting the visibilities of the functions listed above to enhance code clarity and maintainability.

Update: *Resolved at commits bdef64a, cbb704d and pull request #118 at commit 9eaa379.*

N-08 State Variable Visibility Not Explicitly Declared

Within `EmissaryLib.sol`, the `NOT_SCHEDULED` constant lacks an explicitly declared visibility.

For improved code clarity, consider always explicitly declaring the visibility of state variables, even when the default visibility matches the intended visibility.

Update: *Resolved at commit 5d45379.*

N-09 Incorrect Boolean Casting in `_isConsumedBy` Nonce Check

In the `_isConsumedBy` function, the return value of the nonce check is derived from an `and` operation, but it is not explicitly cast to a boolean. While the Solidity compiler currently inserts opcodes to handle this implicitly, this behavior is not guaranteed across versions, which may result in inconsistent or incorrect evaluations.

Consider explicitly casting the result of the `and` operation to a boolean using `iszero(iszero(...))` to ensure consistent behavior across different compiler versions and maintain clarity of intent.

Update: *Resolved in pull request #125 at commit 156d1be.*

N-10 Gas-Optimization Opportunities

The EVM charges gas for every opcode and for each extra byte of deployed bytecode. Re-using already validated inputs, removing redundant masking, and combining bitwise operations reduce both deployment cost and run-time fees. The points below highlight instances where opportunities for such optimization are available:

- In `_setReentrancyLockAndStartPreparingPermit2Call`, `lockTag` is masked to 12 bytes with `shl(160, shr(160, calldataload(0xa4)))`, even though the compiler reverts if the external call supplies anything other than 12 bytes.
- In `_preprocessAndPerformInitialNativeDeposit`, the expression `firstUnderlyingTokenIsNative := iszero(shr(96, shl(96, calldataload(permittedOffset))))` can be replaced by a single `shl` followed by `iszero`, matching the pattern that is used in other places.
- In `_depositBatchAndRegisterViaPermit2`, computing `idsHash := keccak256(add(ids, 0x20), shl(5, add(totalTokensLessInitialNative, firstUnderlyingTokenIsNative)))` duplicates work already performed in `_preprocessAndPerformInitialNativeDeposit`, where the length is stored for `ids` variable. `idsHash := keccak256(add(ids, 0x20), shl(5, mload(ids)))` is equivalent and cheaper.
- In `setNativeTokenBenchmark`, the condition `if or(iszero(eq(callvalue(), 2)), iszero(iszero(balance(target)))) { … }` performs a redundant `iszero` twice. Using `balance(target)` directly removes an opcodes.

- In `setNativeTokenBenchmark` `if or(or(iszero(success1), iszero(success2)), $SecondCondition) { … }` is shorter as `if or(iszero(and(success1, success2)), $SecondCondition) { … }`.
- `setERC20TokenBenchmark` uses two `call` instructions to detect whether the token address was cold. Any cheaper `EXT*` or `BALANCE` opcode can also warm an account and calculate the gas difference for cold to warm transition.
- In `withdraw`, `call(div(gas(), 2), …)` can be rewritten as `call(shr(1, gas()), …)`, thereby possibly saving 2 gas.
- In `beginPreparingBatchDepositPermit2Calldata`, `end` can be replaced with `tokenChunk`.
- In `verifyAndProcessComponents`, `updatedSpentAmount` can be replaced with `spentAmount += amount` while detecting an overflow if `spentAmount < amount`.
- In `assignEmissary`, `_assignableAt` can be reused in lines 108-109.
- In the `EmissaryAssignmentScheduled` event, indexing `assignableAt` adds gas costs (~375 gas) without improving log searchability.
- The `callAuthorizeClaim` function's check for an excessively large `idsAndAmounts` array is performed after a gas-intensive for loop iterates through it, rendering the check ineffective. Instead, the check should be performed before the loop.

Consider applying the above refactorings to remove redundant bitwise masking, replace multi-step expressions with single opcodes, eliminate dead-code gas checks, and drop superfluous indexed event parameters, thereby reducing byte-code size and transaction fees while improving readability for future audits.

Update: *Resolved in pull request #124 at commit a0ac455.*

N-11 EOAs Can Be Registered as Allocators

In the protocol, allocators are expected to implement specific logic, such as the `authorizeClaim` function, which is invoked during claim or allocation processing. These allocators are usually contracts that contain the necessary logic to return a valid authorization signature. However, externally owned accounts (EOAs) lack the capability to respond to such function calls.

The `_registerAllocator` function does not verify whether the allocator address contains contract code, allowing EOAs to be registered. If an EOA is registered as an allocator, and a sponsor initiates a claim or allocated transfer, the protocol will attempt to call the `authorizeClaim` function on the EOA. Since EOAs cannot handle this call, the operation will revert with `InvalidAllocation(allocator)`.

Consider updating `_registerAllocator` to include checks that ensure the allocator address contains contract code and properly implements the `IAllocator` interface when no `proof` is provided. This will help prevent EOAs and contracts that do not adhere to the interface from being registered.

Update: *Acknowledged, not resolved.*

N-12 Typographical Errors

The following typographical errors were identified across the codebase:

- "idiosyncracies" should be "idiosyncrasies".
- "dirtieed" [1, 2] should be "dirtied".
- "by the by the" should be "by the".
- "Ensure sure initial" should be "Ensure initial".

Consider fixing the above typographical mistakes.

Update: *Resolved in pull request #116 at commit 55f3e2b and pull request #119 at commit 53b49c4.*

N-13 Incorrect or Incomplete Documentation

Throughout the codebase, multiple instances of missing or misleading documentation were identified:

- In `EIP712Types.sol`, the comment in line 131 appears to be a legacy artifact.
- The literal `0x9f608b8a` values [1, 2, 3, 4, 5, 6] in `TransferBenchmarkLib.sol` are not commented to be matching the `InvalidBenchmark` error.
- The Natspec of the `_scheduleEmissaryAssignment` and `_assignEmissary` function mentions to use the `toAllocatorIdIfRegistered` function, which is not the case.
- The Natspec of the `_prepareIdsAndGetBalances` function states that the `ids` have to be provided in ascending order. This could be also stated in the external functions `batchDepositViaPermit2` and `batchDepositAndRegisterViaPermit2`.
- The `deriveAndWriteWitnessHash` function is described as a `pure` function, although it is a `view` function.
- The `DOMAIN_SEPARATOR` function is described as a `pure` function, although it is a `view` function.

- The `usingMultichainClaimWithWitness` function is used in `_toMultichainClaimWithWitnessMessageHash` instead of `ClaimHashLib.toMessageHashes` as documented.
- The `usingExogenousMultichainClaimWithWitness` function is used in `_toExogenousMultichainClaimWithWitnessMessageHash` instead of `ClaimHashLib.toMessageHashes` as documented.
- The `usingExogenousMultichainClaimWithWitness` function is used in `_toExogenousMultichainClaimWithWitnessMessageHash` instead of `ClaimHashLib._toMultichainClaimWithWitnessMessageHash` as documented.
- The `emissary` argument of the `assignEmissary` function is not documented with a dedicated `@param` tag.
- The `depositor` argument of the `depositERC20ViaPermit2` function is not documented with a dedicated `@param` tag.
- The `idsAndAmounts` argument of the `batchDepositAndRegisterFor` function is described as "The address of the ERC20 token to deposit", although it is an array of 6909 token IDs and amount tuples.
- In the `depositNativeAndRegisterFor` function, the NatSpec comment indicates that the amount of the claim must be explicitly provided to ensure that the correct claim hash is derived. However, the function signature does not include a parameter for an explicit claim amount.
- The line comment given in the `scheduleEmissaryAssignment` function mentions that "five bit `resetPeriod` from `lockTag`" is extracted during the execution, although the `resetPeriod` consists of 3 bits only.

Consider correcting and adding additional documentation to ease the reasoning of the codebase.

Update: *Resolved in pull request #133 at commit 88ea771.*

N-14 Unfinished Implementation Notes in Codebase

In several parts of the codebase, there are inline comments indicating potential improvements or alternative implementations that have not been acted upon. These comments appear to be placeholders for future decisions rather than finalized design choices. While they may be useful during development, they can cause ambiguity for maintainers, auditors, and contributors who are unsure whether the current implementation is intentional or still under evaluation.

Examples include:

- A note in `TransferBenchmarkLib.sol`, line 109 about potentially using `TSTORE`.
- A remark in `AllocatorLogic.sol`, line 36 questioning the need for an allocator registration check.
- A suggestion in `HashLib.sol`, lines 451–452 to possibly refactor using two loops.
- A comment in `IdLib.sol`, line 163 proposing a possible `SLOAD` bypass.
- A note in `DepositViaPermit2Logic.sol`, lines 469–471 discussing memory allocation considerations tied to later logic.

Consider reviewing each of the above-listed comments and either implementing the suggested changes or removing the comments if no action is needed.

Update: *Resolved at commit 41718c4 and commit 032fa50.*

N-15 Naming Suggestions

Throughout the codebase, multiple opportunities for better naming were identified:

- The `performBatchTransfer` function is the batch equivalent of the `processTransfer` function. To maintain consistency, consider renaming it to *process_ instead of _perform*.
- The `_buildIdsAndAmounts` function also checks for consistent allocator IDs. Consider reflecting this behavior in the function name.

Consider applying the above suggestions to improve the clarity and maintainability of the codebase.

Update: *Resolved in pull request #126 at commit b73738f.*

N-16 Improper Typecasting in `EfficiencyLib` Functions

The `EfficiencyLib` library includes utility functions intended for converting values to and from different types, such as `asBool`, `asBytes12`, and various versions of `asUint256`. In low-level operations, especially when using inline assembly, it is crucial to ensure that type conversions do not leave residual bits—commonly referred to as "dirty bits"—which may lead to unintended behavior if the values are later reused or cast back to other types.

The `asBool` function uses inline assembly to cast a `uint256` value to a `bool`. However, if the original `uint256` value contains any bits set beyond the least significant bit, these bits will remain after the conversion. If the `bool` is later cast back to `uint256`, it may retain the original, unclean value instead of a proper `0` or `1`. Similarly, the `asBytes12` function does not apply a masking operation, which may result in leftover bits from the original value being preserved in the 12-byte output.

Consider updating the `asBool` implementation to explicitly cast the value using the `iszero` opcode or by applying a logical comparison to ensure a clean `0` or `1` output. For `asBytes12`, apply a bitwise `and` operation with a `bytes12` mask to ensure that only the upper 12 bytes are retained. These changes will prevent the propagation of unintended data through dirty bits.

Update: *Resolved in pull request #123 at commit 3de0ba2.*

N-17 Code Simplifications

Throughout the codebase, multiple opportunities for code simplification were identified:

- The `hasConsumedAllocatorNonce` function of `TheCompact` contract could call `isConsumedByAllocator` directly instead of going through `_hasConsumedAllocatorNonce` and `hasConsumedAllocatorNonce`, which are otherwise not used.
- The `toMessageHashes` functions [1, 2] could invoke the respective private function's logic directly, as done with the other `toMessageHashes` functions, instead of propagating through an extraneous `private` function.
- The `_revertWithInvalidBatchAllocationIfError` function is only used once and its logic could be moved into the `_buildIdsAndAmounts` function which is in line with the other `errorBuffer`-`revert` patterns.
- The first `_validateAllocator` function can be merged with the second `_validateAllocator` function.
- The `toRegisteredAllocatorWithConsumed` function is only used once and internally extracts a `uint96 allocatorId` from a `uint256 id`, performing the same logic as `fromRegisteredAllocatorIdWithConsumed`, making the abstraction redundant. Refactor to call `fromRegisteredAllocatorIdWithConsumed` directly by extracting `allocatorId` and remove the unnecessary `toRegisteredAllocatorWithConsumed`.

Consider implementing the above-listed refactoring suggestions to reduce the code footprint and call path complexity.

Update: *Partially resolved at pull request #121 at commit 3d8b7e4.*

N-18 Inaccuracies in Gas Benchmarking Logic

The `TransferBenchmarkLib` and `TransferLib` libraries contain inaccuracies in their gas benchmarking logic for ERC-20 token transfers. These benchmarks are used by the `ensureBenchmarkExceeded` function to determine whether a fair amount of gas has been provided to the `transfer` call. Imprecise measurements can dilute the meaning of the reference benchmark value.

The following inaccuracies were identified:

- The gas cost of the measured `transfer` call includes gas for its preceding `mstore` operations, since `gas` is sampled before the calldata preparation.
- Additional gas costs are also included in the `transfer` measurement after the call for evaluating the success and return has been performed.
- Similarly, the gas costs of the operations performed between the token transfer attempt and the benchmark check introduce a discrepancy in the measurement.

While the benchmark values can be interpreted as ballpark values, consider adjusting the gas samplings pointed out above to ensure that measurements are taken directly before and after the respective external `call` instructions.

Update: *Partially resolved in pull request #122 at commit 3c973ec.*

N-19 Inconsistent Argument Value Usage in `processBatchClaimWithSponsorDomain`

The `processBatchClaimWithSponsorDomain` function passes `0x140` directly to `processClaimWithBatchComponents`, while other functions use `uint256(0x140).asStubborn()`.

Consider being consistent in how the `0x140` value is passed. Alternatively, consider adding a comment to clarify the reason for passing the plain value.

Update: *Resolved at commit 41f7990.*

Conclusion

The Compact protocol implements an on-chain, ownerless mechanism for locking ERC-20 or native tokens. These tokens can be claimed through an arbiter with the help of an allocator, provided that the claim-specific conditions are met. While the overall design is sound and the code is generally well-structured and documented, multiple high- and medium-severity issues were identified that warrant immediate attention. Most noteworthy are the high-severity issues:

- In a multichain claim scenario, compact verification details are bypassed.
- Emissary configuration is computed improperly due to incorrect memory management in associated logic.

Given that the code is highly optimized due to the extensive use of inline assembly, the Uniswap team is encouraged to further strengthen the test suite based on the issues raised in this report. Comprehensive testing will help ensure that unexpected on-chain behavior does not compromise the safety or liveness of locked funds, in accordance with the trust assumptions.

The Uniswap team is appreciated for being very helpful and responsive throughout the audit, providing prompt clarifications on design decisions and assumptions.