Uniswap: The Compact Security Review

Auditors

Desmond Ho, Lead Security Researcher

Kaden, Security Researcher

Philogy, Security Researcher

Report prepared by: Lucas Goiriz

July 14, 2025

Contents

1 About Spearbit **2**

2 Introduction **2**

3 Risk classification **2**
3.1 Impact . . . 2
3.2 Likelihood . . . 2
3.3 Action required for severity levels . . . 2

4 Executive Summary **3**

5 Findings **4**
5.1 Critical Risk . . . 4
5.1.1 Incorrect storage slot derivation breaks authorization . . . 4
5.2 High Risk . . . 6
5.2.1 `MultichainCompact` and `BatchCompact` incompatible with ERC712 due to incorrect hashing . 6
5.3 Medium Risk . . . 7
5.3.1 Attacker can bypass reentrancy lock to double-spend deposit . . . 7
5.3.2 Incorrect argument order for `shr` operation will likely cause benchmarking to revert . . . 10
5.3.3 `TransferBenchmarkLib` Understates Native & ERC20 Transfer Costs . . . 10
5.4 Low Risk . . . 11
5.4.1 Incorrect event signature . . . 11
5.4.2 Empty witness case isn't correctly handled . . . 12
5.5 Gas Optimization . . . 15
5.5.1 Efficient zero value checks . . . 15
5.5.2 Use the same signature offset for batch and non-batch permit2 calls . . . 15
5.5.3 Duplicate addition overflow check . . . 16
5.5.4 Sub-Optimal Reentrancy Lock On Pre-Cancun Chains . . . 17
5.5.5 Sequential Calldata copied to memory using loop instead of ‘calldata . . . 17
5.5.6 Redundant XOR . . . 18
5.5.7 Redundant Shift When Sanitizing Calldata . . . 18
5.5.8 More efficient way to use scratch space . . . 18
5.6 Informational . . . 19
5.6.1 Typos, Minor Recommendations, Redundancies . . . 19
5.6.2 `toAllocatorIdIfRegistered()` doesn't revert for the null address . . . 22
5.6.3 Single native deposits allow 0 amount unlike its ERC20 counterparts . . . 22
5.6.4 Interest rebasing tokens will have interest locked in the contract . . . 23
5.6.5 Lack of signature replay protection may lead to unexpected effects . . . 23
5.6.6 `burn()` doesn't revert for 0 amounts . . . 24
5.6.7 For-loop instead of nested-function for early return can improve readability . . . 25
5.6.8 Conflicting Storage Layout Convention may lead to unsuspecting collisions . . . 28
5.6.9 Use of Unallocated Free Pointer Across Scopes . . . 29
5.6.10 Can Increase Readability by Reordering Struct Fields & Inlining Small Functions . . . 29

1 About Spearbit

Spearbit is a decentralized network of expert security engineers offering reviews and other security related services to Web3 projects with the goal of creating a stronger ecosystem. Our network has experience on every part of the blockchain technology stack, including but not limited to protocol design, smart contracts and the Solidity compiler. Spearbit brings in untapped security talent by enabling expert freelance auditors seeking flexibility to work on interesting projects together.

Learn more about us at spearbit.com

2 Introduction

The Compact is an ownerless ERC-6909 smart contract that lets users deposit tokens into reusable “resource locks” and create conditional agreements (“compacts”) with third parties to transfer or withdraw them once specified conditions are met — designed to support multichain use cases such as cross-chain swaps and escrow.

Disclaimer: This security review does not guarantee against a hack. It is a snapshot in time of Uniswap: The Compact according to the specific commit. Any modifications to the code will require a new security review.

3 Risk classification

Severity level	**Impact: High**	**Impact: Medium**	**Impact: Low**
Likelihood: high	Critical	High	Medium
Likelihood: medium	High	Medium	Low
Likelihood: low	Medium	Low	Low

3.1 Impact

- 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.

3.2 Likelihood

- High - almost certain to happen, easy to perform, or not easy but highly incentivized
- Medium - only conditionally possible or incentivized, but still relatively likely
- Low - requires stars to align, or little-to-no incentive

3.3 Action required for severity levels

- Critical - Must fix as soon as possible (if already deployed)
- High - Must fix (before deployment if not already deployed)
- Medium - Should fix
- Low - Could fix

4 Executive Summary

Over the course of 29 days in total, Uniswap Labs engaged with Spearbit to review the the-compact protocol. In this period of time a total of **25** issues were found.

Summary

Project Name	Uniswap Labs
Repository	the-compact
Commit	102fa069
Type of Project	DeFi, AMM
Audit Timeline	Apr 30th to May 29th

Issues Found

Severity	**Count**	**Fixed**	**Acknowledged**
Critical Risk	1	1	0
High Risk	1	1	0
Medium Risk	3	3	0
Low Risk	2	2	0
Gas Optimizations	8	8	0
Informational	10	10	0
Total	**25**	**25**	**0**

5 Findings

5.1 Critical Risk

5.1.1 Incorrect storage slot derivation breaks authorization

Severity: Critical Risk

Context: EmissaryLib.sol#L65-L84

Summary: The derivation for the emissary configuration storage slot fails to include the `sponsor` which is necessary for authorization. As a result, attackers can arbitrarily set the emissaries of users, allowing them to steal funds from their resource locks by processing malicious claims.

Finding Description: In `EmissaryLib._getEmissaryConfig`, we compute the storage slot for the configuration for a given emissary. This is done by closely packing the `sponsor`, `_EMISSARY_SCOPE`, and `lockTag` and hashing them to derive the storage slot to use:

```
function _getEmissaryConfig(address sponsor, bytes12 lockTag)
    private
    pure
    returns (EmissaryConfig storage config)
{
    assembly ("memory-safe") {
        // Pack data for computing storage slot.
        mstore(0x14, sponsor) // Offset 0x14 (20 bytes): Store 20-byte sponsor address
        mstore(0, _EMISSARY_SCOPE) // Offset 0 (0 bytes): Store 4-byte scope value
        mstore(0x20, lockTag) // Offset 0x20 (12 bytes): Store 12-byte lock tag

        // Compute storage slot from packed data.
        // Start at offset 0x1c (28 bytes), which includes:
        // - The 4 bytes of _EMISSARY_SCOPE (which is stored at position 0x00)
        // - The entire 20-byte sponsor address (which starts at position 0x14)
        // - The entire 12-byte lock tag (which starts at position 0x34)
        // Hash 0x24 (36 bytes) of data in total
        config.slot := keccak256(0x1c, 0x24)
    }
}
```

However, the problem with this logic is that while placing everything in memory, we overwrite the `sponsor` with the `locktag`. This occurs because we place the `sponsor` at 0x14 such that the relevant bytes of the address start at 0x20, then we write to 0x20 again with the `lockTag`, completely overwriting the `sponsor`. We can take a look at the resulting memory in `chisel`:

```
 uint256 private constant _EMISSARY_SCOPE = 0x2d5c707;
 bytes12 lockTag = 0xffffffffffffffffffffffff
 address sponsor = 0x95222290dd7278aa3ddd389cc1e1d165cc4bafe5
 assembly {
    mstore(0x14, sponsor) // Offset 0x14 (20 bytes): Store 20-byte sponsor address
    mstore(0, _EMISSARY_SCOPE) // Offset 0 (0 bytes): Store 4-byte scope value
    mstore(0x20, lockTag) // Offset 0x20 (12 bytes): Store 12-byte lock tag
}
 !memdump
[0x00:0x20]: 0x0000000000000000000000000000000000000000000000000000000002d5c707
[0x20:0x40]: 0xffffffffffffffffffffffff0000000000000000000000000000000000000000
[0x40:0x60]: 0x0000000000000000000000000000000000000000000000000000000000000080
```

As we can see in the above example, the resulting memory does not contain the `sponsor` address.

Impact Explanation: We pass the `msg.sender` as the `sponsor` parameter so that users can only manipulate their own emissary configuration. Effectively, the `sponsor` being included in the slot derivation is an authorization

mechanism. As a result, the `sponsor` not being included in the slot derivation allows anyone to arbitrarily set the emissary configuration of another user.

The emissary is an important role in the system as it has the power to sign a claim on behalf of the sponsor. Thus, by exploiting the unsafe emissary derivation, an attacker can execute claims on behalf of any user to steal funds from their resource locks.

Likelihood Explanation: Since this incorrect slot derivation applies to every sponsor, every sponsor would be vulnerable to this attack and thus every open resource lock could potentially be stolen. The only resource locks which could be protected would be those with already assigned emissaries, in which case an attacker would have to schedule a new emissary assignment, and if noticed, anyone could protect that lock by re-scheduling until the lock can be force withdrawn.

Proof of Concept: The following test should pass as different users should be assigning different emissaries with the same `lockTag`. However, this fails with `EmissaryAssignmentUnavailable` because they are each assigning to the same emissary configuration slot.

```
function test_differentPeopleAssigningEmissary() public {
  // Setup: register allocator
  ResetPeriod resetPeriod = ResetPeriod.TenMinutes;
  Scope scope = Scope.Multichain;

  vm.prank(allocator);
  uint96 allocatorId = theCompact.__registerAllocator(allocator, "");

  bytes12 lockTag =
      bytes12(bytes32((uint256(scope) << 255) | (uint256(resetPeriod) << 252) | (uint256(allocatorId)
      ↪  << 160)));

  // Create 2 emissaries
  address emissary1 = address(new AlwaysOKEmissary());
  address emissary2 = address(new AlwaysOKEmissary());

  // Assign emissary
  vm.prank(swapper);
  theCompact.assignEmissary(lockTag, emissary1);

  // Verify: emissary status of swapper
  (EmissaryStatus status, uint256 emissaryAssignableAt, address currentEmissary) =
      theCompact.getEmissaryStatus(swapper, lockTag);
  assertEq(uint256(status), uint256(EmissaryStatus.Enabled), "Status should be enabled");
  assertEq(emissaryAssignableAt, type(uint96).max, "AssignableAt should be max uint96");
  assertEq(currentEmissary, emissary1, "Current emissary should be emissary1");

  // Different swapper assigning a different emissary
  address swapper2 = makeAddr("swapper2");
  vm.prank(swapper2);
  theCompact.assignEmissary(lockTag, emissary2);

  // Verify: emissary status of swapper2
  (status, emissaryAssignableAt, currentEmissary) = theCompact.getEmissaryStatus(swapper2, lockTag);
  assertEq(uint256(status), uint256(EmissaryStatus.Enabled), "Status should be enabled");
  assertEq(emissaryAssignableAt, type(uint96).max, "AssignableAt should be max uint96");
  assertEq(currentEmissary, emissary2, "Current emissary should be emissary2");
}
```

Recommendation: We can resolve this by adjusting the way that we pack the relevant values in memory to be provided to derive the storage slot, taking care to ensure that we are respecting memory safety in the process.

This can be accomplished first by defining `_EMISSARY_SCOPE` as a `bytes4` constant, leading to it being right padded with zero bytes:

```
- uint256 private constant _EMISSARY_SCOPE = 0x2d5c707; // Note that this wasn't a full 4 bytes, hence
↪  the `e` added below
+ bytes4 private constant _EMISSARY_SCOPE = 0x2d5c707e;
```

Then this allows us to adjust the way we place memory such that we don't have overlapping bytes, while still staying within scratch space, adjusting the range to hash accordingly:

```
  assembly ("memory-safe") {
      // Pack data for computing storage slot.
-     mstore(0x14, sponsor) // Offset 0x14 (20 bytes): Store 20-byte sponsor address
-     mstore(0, _EMISSARY_SCOPE) // Offset 0 (0 bytes): Store 4-byte scope value
-     mstore(0x20, lockTag) // Offset 0x20 (12 bytes): Store 12-byte lock tag
+     mstore(0x00, _EMISSARY_SCOPE) // Offset 0 (0 bytes): Store 4-byte scope value
+     mstore(0x04, sponsor) // Offset 0x10 (16 bytes): Store 20-byte sponsor address
+     mstore(0x24, lockTag) // Offset 0x24 (36 bytes): Store 12-byte lockTag

      // Compute storage slot from packed data.
-     // Start at offset 0x1c (28 bytes), which includes:
+     // Start at offset 0x00 (0 bytes), which includes:
      // - The 4 bytes of _EMISSARY_SCOPE (which is stored at position 0x00)
-     // - The entire 20-byte sponsor address (which starts at position 0x14)
-     // - The entire 12-byte lock tag (which starts at position 0x34)
-     // Hash 0x24 (36 bytes) of data in total
-     config.slot := keccak256(0x1c, 0x24)
+     // - The entire 20-byte sponsor address (which starts at position 0x10)
+     // - The entire 12-byte lock tag (which starts at position 0x24)
+     // Hash 0x30 (48 bytes) of data in total
+     config.slot := keccak256(0x00, 0x30)
  }
```

Uniswap Labs: Fixed in commit 3caea0b7.

Spearbit: Fix verified.

5.2 High Risk

5.2.1 `MultichainCompact` and `BatchCompact` incompatible with ERC712 due to incorrect hashing

Severity: High Risk

Context: HashLib.sol#L527-L543, HashLib.sol#L556-L584, HashLib.sol#L591-L620

Description: In order to work with major wallets like MetaMask the Compact aims to be ERC712 compatible so that users can easily & direct sign messages for the contract. However due to incorrect handling of `idsAndAmounts: uint256[2][]` across the two types they are not compatible.

Specifically in the ERC712 standard it is specified that to encode arrays for the struct hash:

> The array values are encoded as the `keccak256` hash of the concatenated `encodeData` of their contents (i.e. the encoding of `SomeType[5]` is identical to that of a struct containing five members of type `SomeType`).

However in both cases the Compact simply hashes the flattened array instead of first hashing the inner `uint256[2]` arrays and then hashing the concatenated list of hashes to derive the commitment of `uint256[2][] idsAndAmounts`.

Impact: While the hashing is incorrect and will not be compatible with major wallets and therefore will be practically unavailable to most users, it is sound in its current state. It is sound in the sense that for custom wallets/signers if they do produce a signature following this non-standard hashing scheme it will not allow for collisions against other message digests, fulfilling the original purposes of the standard. This is why the impact has been deemed

"Medium" is it makes major functionality practically inaccessible but does not fundamentally allow the theft of or destruction of user or protocol funds.

Recommendation: Fix the hashing implementations or redefine the type to just use a flat `uint256[]` array enforcing that the length is a multiple of 2.

Uniswap Labs: Fixed in commit cda33d86.

Spearbit: Fix verified.

5.3 Medium Risk

5.3.1 Attacker can bypass reentrancy lock to double-spend deposit

Severity: Medium Risk

Context: ConstructorLogic.sol#L66-L89

Description: To correctly account for a variety of tokens including fee-on-transfer the Compact measures the contract's balance before and after a transfer in to determine e.g. in `src/lib/DirectDepositLogic.sol`:

```
function _transferAndDeposit(address token, address to, uint256 id, uint256 amount)
    private
    returns (uint256 mintedAmount)
{
    // Retrieve initial token balance of this contract.
    uint256 initialBalance = token.balanceOf(address(this));

    // Transfer tokens from caller to this contract.
    token.safeTransferFrom(msg.sender, address(this), amount);

    // Compare new balance to initial balance and deposit ERC6909 tokens to recipient.
    return _checkBalanceAndDeposit(token, to, id, initialBalance);
}
```

This function is prone to double-counting deposits if you're able to re-enter the function and trigger the deposit again in between balance updates:

```
deposit_1(amount: 20) {
    initialBalance = 100.0

    safeTransferFrom(..., 20) -> reenter deposit_2(amount: 20) {
        initialBalance = 100.0
        safeTransferFrom(..., 20)
        balanceAfter = 120.0

        credit_depositor(120.0 - 100.0 = 20.0)
    }

    balanceAfter = 140.0
    credit_depositor(140.0 - 100.0 = 40.0)

}

Total new balance:  60
Actually deposited: 40
```

The reentrancy lock is meant to prevent this, however it has a flaw that allows for a one-time bypass under specific circumstances. Note how the reentrancy lock doesn't use normal/transient storage directly but instead the `_setTstorish` & `_clearTstorish` methods. These are meant to use transient storage but default to normal storage

if it detects that it's running in an EVM version that doesn't support transient storage yet (from the constructor in `lib/tstorish/src/Tstorish.sol`):

```
// Determine if TSTORE is supported.
bool tstoreInitialSupport = _testTload(tloadTestContract);

if (tstoreInitialSupport) {
    // If TSTORE is supported, set functions to their versions that use
    // tstore/tload directly without support checks.
    _setTstorish = _setTstore;
    _getTstorish = _getTstore;
    _clearTstorish = _clearTstore;
} else {
    // If TSTORE is not supported, set functions to their versions that
    // fallback to sstore/sload until _tstoreSupport is true.
    _setTstorish = _setTstorishWithSstoreFallback;
    _getTstorish = _getTstorishWithSloadFallback;
    _clearTstorish = _clearTstorishWithSstoreFallback;
}
```

If a chain starts off by not having transient storage support (pre-Cancun EVM version) but later upgrades to one that does then the `__activateTstore` method can be used to get the tstorish methods to use the more efficient transient storage:

```
function __activateTstore() external {
    // Ensure this function is triggered from an externally-owned account.
    if (msg.sender != tx.origin) {
        revert OnlyDirectCalls();
    }

    // Determine if TSTORE can potentially be activated.
    if (_tstoreInitialSupport || _tstoreSupport) {
        revert TStoreAlreadyActivated();
    }

    // Determine if TSTORE can be activated and revert if not.
    if (!_testTload(_tloadTestContract)) {
        revert TStoreNotSupported();
    }

    // Mark TSTORE as activated.
    _tstoreSupport = true;
}
```

This is problematic because if activation happens *during* a call that's using the reentrancy lock it'll change the slot it uses to check whether the lock was engaged from normal storage to transient storage:

```
deposit_1 {
    // Reentrancy lock initial check & lock
    require(sload(LOCK_SLOT) == 0);
    sstore(LOCK_SLOT, LOCKED_VALUE)

    // ...execute deposit logic

    ... -> attacker scope / contract {
        call __activateTstore()
        deposit_2 {
            // Inner: Reentrancy lock initial check & lock
            //  Passes because lock now uses *transient* storage and slots default to 0
            require(tload(LOCK_SLOT) == 0);
            tstore(LOCK_SLOT, LOCKED_VALUE)

            // ...execute deposit logic

            // Inner: Reentrancy lock unlock
            tstore(LOCK_SLOT, 0);
        }
    }

    // Reentrancy lock unlock
    tstore(LOCK_SLOT, 0);
}
```

An attacker can now use this reentrancy exploit to achieve a double-counting of their deposit. While they can only do this once as `__activateTstore` is a one-time state transition they can double-spend any tokens they like using the `batchDeposit` method. If none of the tokens they want to double-spend support callbacks the attacker can get access to control flow by adding their own dummy token to the batch.

Likelihood Explanation: While the impact of this vulnerability is quite severe, allowing an attacker to perform a double-spend allowing them to drain as many tokens as they can get liquidity for the likelihood diminishes the severity. The likelihood is deemed **Low** because it relies on particular conditions:

1. `TheCompact` must be deployed on EVM chains that do not run the latest (Prague) or even last EVM. version (Cancun).
2. The attacker must perform this attack before an honest actor triggers `__activateTstore` themselves.
3. When upgrading, the chain must skip at least one upgrade going from Pre-Cancun to Prague or above so that the attacker can leverage EIP-7702 to bypass the `msg.sender != tx.origin` check.

- Condition 1. is the one that impacts the likelihood the most as very few chains are still running pre-Cancun versions.
- Condition 2. is quite likely, if this exploit weren't known there would be no reason to think that honest actors would be pressured to trigger `__activateTstore`, this is unlike an attacker who upon discovering the attack would have a significant incentive to be the first to execute it as soon as an upgrade takes place.
- Condition 3. is also relatively likely, it is quite conceivable that a chain that is as out-dated as to run a pre-Cancun version would directly upgrade to Prague and skip a separate upgrade event.

Recommendation: Either remove the activation mechanism or ensure that activation registers a block number *after* which transient storage is to be used, this will ensure that the switch from transient to persisent storage is consistent and atomic, happening implicitly between transactions.

Uniswap Labs: Fixed in commit be468425.

Spearbit: Fix verified.

5.3.2 Incorrect argument order for `shr` operation will likely cause benchmarking to revert

Severity: Medium Risk

Context: TransferBenchmarkLib.sol#L48

Description: The argument order for the `shr` function is incorrect; causing the number `96` to be shifted by the hash value, rather than shifting the hash value by 96 bits. As such, it is likely for the calculated `target` to be the null address, which holds non-zero native balance for most chains. This would cause the benchmarking to fail because of the requirement for the target to have zero native balance.

Recommendation:

```
- let target := shr(keccak256(0x0c, 0x34), 96)
+ let target := shr(96, keccak256(0x0c, 0x34))
```

Uniswap Labs: Fixed in commit 90f0fce5.

Spearbit: Fix verified.

5.3.3 `TransferBenchmarkLib` Understates Native & ERC20 Transfer Costs

Severity: Medium Risk

Context: TransferBenchmarkLib.sol#L46-L80, TransferBenchmarkLib.sol#L115-L176

Description:

- **Native Transfer:**
  1. The call to `balance(target)` warms the `target` account in advance meaning the two following measurements are for warm accounts.
  2. The first measurement will always yield a larger cost on all major EVM versions because a native ETH transfer to an empty account bears a larger cost (31.8k warm) than when the account is not empty (6.8k warm) making the comparison redundant.
  3. Even if the checks measured the expected values a direct strict comparison `firstCost <= secondCost` is not guaranteed to give the correct result as there can be slight differences in base cost based on what auxiliary instructions the compiler generates between the two `gas()` invocations (`SWAPN`, `DUPN`, etc.)...
- **ERC20 Transfer:**
  1. Similar to *Native Transfer* it also relies upon direct comparison.
  2. Does not measure the worst case, it only tries to check whether the token account is warm, however it does not attempt to measure & correct for recipient/sender balance slots being warm.
  3. The `BenchmarkERC20` token being used as a benchmark is very optimized, using a notoriously efficient Solady `ERC20` implementation as its foundation, compiled with the latest solc compiler, with no upgradeable proxy overhead or other on-transfer hook mechanisms, this is unlike common tokens like USDC / USDT that are upgradeable proxies or many memecoins that have additional gas overhead on transfers due to certain mechanisms.

Impact: To ensure withdrawals are not DoSed by excessive gas costs from recipients/tokens the `TransferLib.withdraw` function falls back to transferring the underlying resource-lock balance directly when transfers fail. However to prevent malicious actors from supplying too little gas to always trigger the fallback it relies on the benchmark to verify that a good-faith attempt was made to supply sufficient gas to process the withdrawal. The benchmark ensures the gas limits are not hardcoded and can adapt to different chains and even changes in gas schedules.

Under-accounting the costs in the transfer benchmark makes it more likely that arbiters or callers thereof can save gas by supplying too little and pushing the costs on recipients.

Likelihood: The likelihood is deemed "High" because at least for the ERC20 benchmark the measurement when the sender & recipient balance slots are warmed is *very* optimistic and it is quite conceivable that a normal token transfer may more than 2x this cost allowing withdrawals to trigger the more expensive fallback.

Recommendation: For Native transfers:

- Ensure the "warmness" of the account is correctly measures by measuring the difference between the first and second call to `balance(target)`.
- When comparing the first and second gas diff for the `balance` call ensure to build in some padding to account for subtle bytecode generation differences e.g. `require(firstDiff + 20 >= secondDiff)`.

For ERC20 transfers:

- Measure the "warmness" of the sender and recipient balances after the "warmness" of the token account is measured by making calls to `balanceOf`.
- Similar to native transfers build in a bit of padding when comparing gas diffs.
- Use a token benchmark implementation that uses an upgradeable-proxy pattern and ensure that the implementation account & slots are cold in the benchmark to account for the more general token case.

Uniswap Labs: Fixed in PR 152.

Spearbit: Fix verified.

5.4 Low Risk

5.4.1 Incorrect event signature

Severity: Low Risk

Context: EventLib.sol#L12-L46

Description: In `EventLib.sol`, we declare the precomputed `_CLAIM_EVENT_SIGNATURE` according to the signature `Claim(address,address,address,bytes32)`:

```
// keccak256(bytes("Claim(address,address,address,bytes32)")).
uint256 private constant _CLAIM_EVENT_SIGNATURE =
↪  0x770c32a2314b700d6239ee35ba23a9690f2fceb93a55d8c753e953059b3b18d4;
```

The problem with this is that when we emit a `Claim` event, we include an additional `nonce` parameter, which is not included in the signature:

```
function emitClaim(address sponsor, bytes32 claimHash, address allocator, uint256 nonce) internal {
    assembly ("memory-safe") {
        // Emit the Claim event:
        //  - topic1: Claim event signature
        //  - topic2: sponsor address (sanitized)
        //  - topic3: allocator address (sanitized)
        //  - topic4: caller address
        //  - data: messageHash, nonce
        mstore(0, claimHash)
        mstore(0x20, nonce)
        log4(
            0,
            0x40,
            _CLAIM_EVENT_SIGNATURE,
            shr(0x60, shl(0x60, sponsor)),
            shr(0x60, shl(0x60, allocator)),
            caller()
        )
    }
}
```

As we can see above, the emitted event includes three address parameters, a bytes32 `messageHash`, and a `nonce`, but the `Claim` event signature only includes the three address parameters and one `bytes32` parameter.

Recommendation: Replace the `_CLAIM_EVENT_SIGNATURE` with a signature including a parameter for the `nonce`:

```
- // keccak256(bytes("Claim(address,address,address,bytes32)")).
- uint256 private constant _CLAIM_EVENT_SIGNATURE =
↪  0x770c32a2314b700d6239ee35ba23a9690f2fceb93a55d8c753e953059b3b18d4;
+ // keccak256(bytes("Claim(address,address,address,bytes32,uint256)")).
+ uint256 private constant _CLAIM_EVENT_SIGNATURE =
↪  0x9a071f16ca19062495c8c0e832e4541b4453cd2995fd631b0b7e9f0ee300ff12;
```

Uniswap Labs: Fixed in commit 1b01b6ae.

Spearbit: Fix verified.

5.4.2 Empty witness case isn't correctly handled

Severity: Low Risk

Context: DepositViaPermit2Lib.sol#L217-L258

Description: For the `no-witness` case, the insertion of the token permissions typestring fragment is in the incorrect offset, causing the resultant string to exclude the opening parenthesis of Mandate, ie. it is `Mandate)` when it should be `Mandate()`. As such, it would be impossible to derive the correct digest for `permit2` signature verification, causing this path to be un-useable and requiring projects to pass in dummy witnesses.

Proof of Concept:

```
string constant compactWithEmptyWitnessTypestring = "Compact(address arbiter,address sponsor,uint256
↪  nonce,uint256 expires,uint256 id,uint256 amount)";
bytes32 constant compactWithEmptyWitnessTypehash = keccak256(bytes(compactWithEmptyWitnessTypestring));

function test_depositAndRegisterWithPermit2EmptyWitness() public {
  // Setup test parameters
  TestParams memory params;
  params.resetPeriod = ResetPeriod.TenMinutes;
  params.scope = Scope.Multichain;
  params.amount = 1e18;
  params.nonce = 0;
  params.deadline = block.timestamp + 1000;

  // Initialize claim
  Claim memory claim;
  claim.sponsor = swapper;
  claim.nonce = params.nonce;
  claim.expires = block.timestamp + 1000;
  claim.allocatedAmount = params.amount;
  claim.witnessTypestring = "";
  claim.sponsorSignature = "";

   // Create domain separator
  bytes32 domainSeparator;
  {
      domainSeparator = keccak256(
          abi.encode(permit2EIP712DomainHash, keccak256(bytes("Permit2")), block.chainid,
          ↪  address(permit2))
      );
      assertEq(domainSeparator, EIP712(permit2).DOMAIN_SEPARATOR());
  }

  // Create witness and id
  LockDetails memory expectedDetails;
```

```
uint96 allocatorId;
{
    // Register allocator and setup
    bytes12 lockTag;
    {
        (allocatorId, lockTag) = _registerAllocator(allocator);
    }
    expectedDetails.lockTag = lockTag;

    claim.witness = keccak256(abi.encode(""));
    claim.id = uint256(bytes32(lockTag)) | uint256(uint160(address(token)));
}

// Create claim hash
bytes32 claimHash;
{
    CreateClaimHashWithWitnessArgs memory args;
    args.typehash = compactWithEmptyWitnessTypehash;
    args.arbiter = 0x2222222222222222222222222222222222222222;
    args.sponsor = claim.sponsor;
    args.nonce = claim.nonce;
    args.expires = claim.expires;
    args.id = claim.id;
    args.amount = claim.allocatedAmount;
    args.witness = claim.witness;

    claimHash = _createClaimHashWithWitness(args);
}

// Create activation typehash and permit signature
bytes memory signature;
ISignatureTransfer.PermitTransferFrom memory permit;
{

    bytes32 activationTypehash = keccak256(
        bytes(
            string.concat("Activation(address activator,uint256 id,Compact compact)",
            ↪  compactWithEmptyWitnessTypestring)
        )
    );

    {
        bytes32 tokenPermissionsHash = keccak256(
            abi.encode(
                keccak256("TokenPermissions(address token,uint256 amount)"), address(token),
                ↪  params.amount
            )
        );

        bytes32 permitWitnessHash;
        {
            bytes32 activationHash =
                keccak256(abi.encode(activationTypehash, address(1010), claim.id, claimHash));

            permitWitnessHash = keccak256(
                abi.encode(
                    keccak256(
                        "PermitWitnessTransferFrom(TokenPermissions permitted,address spender,uint256
                        ↪  nonce,uint256 deadline,Activation witness)Activation(address
                        ↪  activator,uint256 id,Compact compact)Compact(address arbiter,address
                        ↪  sponsor,uint256 nonce,uint256 expires,uint256 id,uint256
                        ↪  amount)TokenPermissions(address token,uint256 amount)"
```

```
                    ),
                    tokenPermissionsHash,
                    address(theCompact), // spender
                    params.nonce,
                    params.deadline,
                    activationHash
                )
            );
        }

        bytes32 digest = keccak256(abi.encodePacked(bytes2(0x1901), domainSeparator,
        ↪  permitWitnessHash));

        bytes32 r;
        bytes32 vs;
        (r, vs) = vm.signCompact(swapperPrivateKey, digest);
        signature = abi.encodePacked(r, vs);
    }

    // Create permit
    {
        permit = ISignatureTransfer.PermitTransferFrom({
            permitted: ISignatureTransfer.TokenPermissions({ token: address(token), amount:
            ↪  params.amount }),
            nonce: params.nonce,
            deadline: params.deadline
        });
    }

    // Setup expectation for permitWitnessTransferFrom call
    {
        bytes32 activationHash = keccak256(abi.encode(activationTypehash, address(1010), claim.id,
        ↪  claimHash));

        vm.expectCall(
            address(permit2),
            abi.encodeWithSignature(
                "permitWitnessTransferFrom(((address,uint256),uint256,uint256),(address,uint256),addr
                ↪  ess,bytes32,string,bytes)",
                permit,
                ISignatureTransfer.SignatureTransferDetails({
                    to: address(theCompact),
                    requestedAmount: params.amount
                }),
                swapper,
                activationHash,
                "Activation witness)Activation(address activator,uint256 id,Compact
                ↪  compact)Compact(address arbiter,address sponsor,uint256 nonce,uint256
                ↪  expires,uint256 id,uint256 amount)TokenPermissions(address token,uint256 amount)",
                signature
            )
        );
    }
}

// Deposit and register
{
    vm.prank(address(1010));
    // reverts due to incorrect digest derivation
    theCompact.depositERC20AndRegisterViaPermit2(
        permit,
        swapper,
```

```
            expectedDetails.lockTag,
            claimHash,
            CompactCategory.Compact,
            "",
            signature
        );
    }
}
```

The test reverts with custom error `0x815e1d64`, which corresponds to `InvalidSigner()` from `permit2`.

Recommendation: Decide on how an empty mandate witness looks like, then appropriately insert the remaining typestring fragments. The same will have to be done for claims, where the ClaimProcessor's `typehash` is also derived.

Uniswap Labs: Fixed in commit c8b7c645.

Spearbit: Fix verified.

5.5 Gas Optimization

5.5.1 Efficient zero value checks

Severity: Gas Optimization

Context: DepositViaPermit2Logic.sol#L374, DirectDepositLogic.sol#L74

Description: In a couple different places, we validate that values of fixed sized types are zero by first sanitizing the values to remove any possible dirty bytes. The pattern we use to sanitize these values is shifting left and shifting right by the amount of unused bytes for the given type, e.g.:

```
firstUnderlyingTokenIsNative := iszero(shr(96, shl(96, id)))
```

Since we only intend to determine whether the value is zero, we can remove the right shift, instead only shifting to the left since any possible dirty bytes have already been removed at this point.

Recommendation: Remove redundant right shift when sanitizing values to determine if they are zero:

```
- firstUnderlyingTokenIsNative := iszero(shr(96, shl(96, id)))
+ firstUnderlyingTokenIsNative := iszero(shl(96, id))
```

Uniswap Labs: Fixed in commit d9769835.

Spearbit: Fix verified.

5.5.2 Use the same signature offset for batch and non-batch permit2 calls

Severity: Gas Optimization

Context: DepositViaPermit2Logic.sol#L306-L314

Description: In `_depositBatchAndRegisterViaPermit2`, we compute the signature offset to be used in constructing memory for our permit2 call:

```
// Declare variable for signature offset value.
uint256 signatureOffsetValue;
assembly ("memory-safe") {
    // Derive the total memory offset for the witness.
    let totalWitnessMemoryOffset :=
        and(
            add(
                add(0x11d, add(witness.length, iszero(iszero(witness.length)))),
                mul(eq(compactCategory, 1), 0x0b)
            ),
            not(0x1f)
        )

    // Derive the signature offset value.
    signatureOffsetValue :=
        add(add(0x180, shl(7, totalTokensLessInitialNative)), totalWitnessMemoryOffset)
}
```

The above snippet handles the case for both a batch or non-batch typestring, hence why we add 0x0b if `eq(compactCategory, 1)`. The difference between these cases is likely to only be one word of memory expansion, with an upper bound of two words, so as long as the memory size isn't very large, it should be more efficient to avoid all this computation and instead provide the larger possible offset for both cases. We can compute the upper bound for the offset (batch case) as follows:

- Batch typestring length: `0x48` (initial activation fragment) + `0x93` (compact fragment) + `0x2f` (token permission fragment) = `0x10a`.
- Add `0x1f` as part of pattern to round up to the nearest word = `0x129`.

So the resulting `totalWitnessMemoryOffset` would be computed as:

```
let totalWitnessMemoryOffset :=
    and(
        add(0x129, add(witness.length, iszero(iszero(witness.length)))),
        not(0x1f)
    )
```

Here we removed: 1 add, 2 push, 1 mul, 1 eq ~= 17 runtime gas. Assuming we expand memory by a maximum of two words (64 bytes), as long as the current memory size is less than ~12000 bytes, this optimization should consume less gas. Since exceeding ~12000 bytes of memory would require a witness >11500 bytes, it's highly likely that we will consume less gas using this pattern.

Recommendation: Replace the `totalWitnessMemoryOffset` computation as indicated above:

```
  let totalWitnessMemoryOffset :=
      and(
-         add(
-             add(0x11d, add(witness.length, iszero(iszero(witness.length)))),
-             mul(eq(compactCategory, 1), 0x0b)
-         ),
+         add(0x129, add(witness.length, iszero(iszero(witness.length)))),
          not(0x1f)
      )
```

Uniswap Labs: Fixed in PR 140.

Spearbit: Fix verified.

5.5.3 Duplicate addition overflow check

Severity: Gas Optimization

Context: HashLib.sol#L62-L81, HashLib.sol#L142-L172

Description: In `toTransferMessageHash()` and `toBatchTransferMessageHash()`, a check exists to ensure that the total amount doesn't have addition overflow. This is a duplicate check, as it was previously performed in `ComponentLib.aggregate()` when `transfer.recipients.aggregate()` / `component.portions.aggregate()` is called respectively.

Recommendation: Consider if this check should be removed, as there may be a use-case where these libraries are used separately, so it may be worthwhile to leave it as it is.

Uniswap Labs: Fixed in commit 26ae1d87.

Spearbit: Fix verified.

5.5.4 Sub-Optimal Reentrancy Lock On Pre-Cancun Chains

Severity: Gas Optimization

Context: ConstructorLogic.sol#L66-L89

Description: `TheCompact` supports EVM versions as far back as Shanghai and does not want to rely on transient storage for its reentrancy lock. It achieves this via the `tstorish/Tstorish.sol` library that provides it an abstraction over persistent/transient storage, falling back to the traditional "storage" when it detects that the EVM version does not support transient storage.

When the reentrancy lock is in the "unlocked" state the slot is set to `0` and to a non-zero value when it's locked. This is inefficient when persistent storage is being used because changing a storage value from `0` to a non-zero value incurs an additional ~20k gas. While this cost is refunded when the slot is reset to `0` to unlock the lock gas refunds are limited to 1/5 of the total gas used in your transaction. This means gas is lost whenever the total gas used is below 100k.

Recommendation: Have the base "unlocked" state be represented by a non-zero value. To maintain compatibility with transient storage who's default state is `0` check whether the lock is "unlocked" by checking whether the slot is `<= 1` this ensures that the gas use is fully optimized for both versions.

Uniswap Labs: Fixed in commit be468425.

Spearbit: Fix verified.

5.5.5 Sequential Calldata copied to memory using loop instead of ‘calldata

Severity: Gas Optimization

Context: DepositViaPermit2Lib.sol#L108-L114

Description: The amounts for the batch permit2 signed transfer are copied from calldata to memory using the same loop that encodes the address.

Recommendation: Use `calldatacopy` to copy the amounts up-front, using the existing loop to overwrite the contents that are not used. This is net more efficient because a `calldatacopy` only incurs a cost of 3-gas per word being copied vs. the mstore which includes more expensive offset calculations:

```
+ calldatacopy(next, add(permittedCalldataLocation, 0x20), end)
  // Iterate through `details` array and copy data from calldata to memory.
  for { let i := 0 } lt(i, end) { i := add(i, 0x40) } {
      // Copy this contract as the recipient address.
      mstore(add(starting, i), address())
-
-     // Copy full token amount as the requested amount.
-     mstore(add(next, i), calldataload(add(permittedCalldataLocation, add(0x20, i))))
  }
```

Uniswap Labs: Fixed in commit 83e76aa4.

Spearbit: Fix verified.

5.5.6 Redundant XOR

Severity: Gas Optimization

Context: EfficiencyLib.sol#L26

Description: The assembly code `xor(account, mul(xor(account, caller()), iszero(account)))` is meant to represent a branch-less more efficient version of `account == address(0) ? msg.sender : account`. However the `xor(account, caller())` is redundant as the `mul` product will only be included (non-zero) if `account == 0`. It's redundant because `xor(0, n) == n`.

Recommendation: Remove the redundant operation:

```
- accountOrCallerIfNull := xor(account, mul(xor(account, caller()), iszero(account)))
+ accountOrCallerIfNull := xor(account, mul(caller(), iszero(account)))
```

Uniswap Labs: Fixed in commit bb4d7fc1.

Spearbit: Fix verified.

5.5.7 Redundant Shift When Sanitizing Calldata

Severity: Gas Optimization

Context: ClaimProcessorLib.sol#L71

Description: The referenced assembly code aims to read an address and clear its upper dirty bits. It does so via two shifts. Illustrated let's imagine the EVM only operated on 2-byte words:

```
calldata: 0x........aabb......
          read value ^^^^

x << shift            -> 0xbb00 (deletes the upper bits from `aa`)
(x << shift) >> shift -> 0x00bb (shifts the value back to the original alignment)
```

Note how the final right-shift essentially clears the lower bytes of the value. We can save on this shift by already reading a left-shifted value. How? We right shift our read:

```
calldata: 0x........aabbXX....
            read value ^^^^

x           -> 0xbbXX (read value is already left shifted)
x >> shift -> 0x00bb (shifts value right, clearing the lower bytes)
```

Recommendation: Apply the described simplification to the relevant code:

```
- sponsor := shr(0x60, shl(0x60, calldataload(add(calldataPointer, 0x40))))
+ sponsor := shr(0x60, calldataload(add(calldataPointer, 0x4c)))
```

Uniswap Labs: Fixed in commit 327d5287.

Spearbit: Fix verified.

5.5.8 More efficient way to use scratch space

Severity: Gas Optimization

Context: WithdrawalLogic.sol#L154-L165

Description: The referenced code does not fully overwrite the free memory pointer. Instead of caching the full memory pointer just to restore it later you can just zero out the written bytes.

Recommendation: Apply the optimization:

```
  function _getCutoffTimeSlot(address account, uint256 id) private pure returns (uint256
  ↪  cutoffTimeSlotLocation) {
      assembly ("memory-safe") {
-         // Retrieve the current free memory pointer.
-         let m := mload(0x40)
-
          // Pack data for computing storage slot.
          mstore(0x14, account)
          mstore(0, _FORCED_WITHDRAWAL_ACTIVATIONS_SCOPE)
          mstore(0x34, id)

          // Compute storage slot from packed data.
          cutoffTimeSlotLocation := keccak256(0x1c, 0x38)

          // Restore the free memory pointer.
-         mstore(0x40, m)
+         mstore(0x34, 0)
      }
  }
```

Uniswap Labs: Fixed in commit 4cfd8ace.

Spearbit: Fix verified.

5.6 Informational

5.6.1 Typos, Minor Recommendations, Redundancies

Severity: Informational

Context: AllocatorLib.sol#L48, AllocatorLib.sol#L99, ConsumerLib.sol#L55, ConsumerLib.sol#L93, DepositViaPermit2Logic.sol#L85, DirectDepositLogic.sol#L80, DomainLib.sol#L34, EmissaryLib.sol#L72-L81, EmissaryLib.sol#L120-L124, EmissaryLib.sol#L167-L170, EmissaryLib.sol#L236, EmissaryLogic.sol#L54-L55, EmissaryLogic.sol#L74-L75, HashLib.sol#L272, HashLib.sol#L552-L553, MetadataLib.sol#L181-L186, TransferBenchmarkLib.sol#L15, TransferBenchmarkLib.sol#L45, TransferLib.sol#L65-L69, TransferLib.sol#L209-L213, TransferLogic.sol#L71, TransferLogic.sol#L118, ValidityLib.sol#L121, EIP712Types.sol#L131, EIP712Types.sol#L182-L183

Description/Recommendation:

- **Typos:**

```
- dirtieed
+ dirtied

- idiosyncracies
+ idiosyncrasies


- valdiate
+ validate

- reentancy
+ reentrancy
```

- **Minor Recommendations:**

```
-   //  * the first token is non-native and the callvalue doesn't equal the first amount
+   //  * the first token is native and the callvalue doesn't equal the first amount


-   // keccak256(_CONSUMER_NONCE_SCOPE ++ account ++ nonce[0:31])
+   // keccak256(_ALLOCATOR_NONCE_SCOPE ++ account ++ nonce[0:31])


    * erroneous hash values. This function also assumes that replacementAmounts.length
-   * does not exceed replacementAmounts.length and will break if the invariant is not
+   * does not exceed idsAndAmounts.length and will break if the invariant is not
    * upheld.


-   // Offset 0 (0 bytes): Store 4-byte scope value
+   // Offset 0 (4 bytes): Store 4-byte scope value

    // Compute storage slot from packed data.
    // Start at offset 0x1c (28 bytes), which includes:
-   // - The 4 bytes of _EMISSARY_SCOPE (which is stored at position 0x00)
+   // - The 4 bytes of _EMISSARY_SCOPE (which starts at position 0x1c)
-   // - The entire 20-byte sponsor address (which starts at position 0x14)
+   // - The entire 20-byte sponsor address (which starts at position 0x20)
    // - The entire 12-byte lock tag (which starts at position 0x34)
    // Hash 0x24 (36 bytes) of data in total


-   // Derive the target for  native token transfer using address.this & salt.
+   // Derive the target for native token transfer using address.this & salt.


-   // uint72(abi.decode(bytes("nt256 id,"), (bytes9)))


-   // uint216(abi.decode(bytes("te mandate)Mandate("), (bytes19)))
+   // uint152(abi.decode(bytes("te mandate)Mandate("), (bytes19)))
-   uint216 constant PERMIT2_ACTIVATION_BATCH_COMPACT_TYPESTRING_FRAGMENT_FIVE =
↪  0x7465206d616e64617465294d616e6461746528;
+   uint152 constant PERMIT2_ACTIVATION_BATCH_COMPACT_TYPESTRING_FRAGMENT_FIVE =
↪  0x7465206d616e64617465294d616e6461746528;


-   *      The function utilizes `toAllocatorIdIfRegistered` to validate and convert the
↪  allocator
-   *      address to its corresponding ID, ensuring that only registered allocators can proceed.
+   *      The function utilizes `hasRegisteredAllocatorId` to validate the allocator ID
+   *      ensuring that only registered allocators can proceed.


-   //


-   // Otherwise, set the provided resetPeriod.
+   // Otherwise, set the provided newEmissary.
```

  - Consider renaming the struct `ExogenousBatchMultichainClaim` to `ExogenousBatchClaim` to match the function name, as is the case with the other claim types.

- **Redundancies:**

  1. Redundant check in `EmissaryLib.extractSameLockTag()`:

```
- // Ensure length is at least 1.
- if (idsAndAmountsLength == 0) {
-     revert InvalidLockTag();
- }
```

because

- for single claims (`claim()`, `multichainClaim()` & `exogenousClaim()`), ID must be explicitly specified and its allocator extracted and validated to be registered.
- for batch claims, (`batchClaim()`, `batchMultichainClaim()` & `exogenousBatchClaim()`), it'll revert when `_buildIdsAndAmounts()` is called.

```
function test_revert_zeroIdsAndAmounts() public {
// Single claims (Claim & MultichainClaim) take in a single ID which must contain a valid
↪  allocator ID
// So just need to test batch claims
BatchClaim memory batchClaim;
batchClaim.expires = block.timestamp + 1000;
vm.prank(swapper);
// reverts with NoIdsAndAmountsProvided
vm.expectRevert(ComponentLib.NoIdsAndAmountsProvided.selector);
theCompact.batchClaim(batchClaim);

BatchMultichainClaim memory batchMultichainClaim;
batchMultichainClaim.expires = block.timestamp + 1000;
vm.prank(swapper);
// reverts with NoIdsAndAmountsProvided
vm.expectRevert(ComponentLib.NoIdsAndAmountsProvided.selector);
theCompact.batchMultichainClaim(batchMultichainClaim);

ExogenousBatchMultichainClaim memory exogenousBatchMultichainClaim;
exogenousBatchMultichainClaim.expires = block.timestamp + 1000;
vm.prank(swapper);
// reverts with NoIdsAndAmountsProvided
vm.expectRevert(ComponentLib.NoIdsAndAmountsProvided.selector);
theCompact.exogenousBatchClaim(exogenousBatchMultichainClaim);
}
```

2. Redundant balance overflow checks.

```
- // Revert on balance overflow.
- if lt(toBalanceAfter, toBalanceBefore) {
-   mstore(0x00, 0x89560ca1) // `BalanceOverflow()`.
-   revert(0x1c, 0x04)
- }
```

If we work with the assumption that the total supply of an ERC6909 ID token is bounded by a native / ERC20 token's supply, it shouldn't ever be possible to have balance overflow, so this check would be redundant. It's a feasible assumption to make; UniV4 assumes a token's supply is capped at `type(uint128).max`.

3. Redundant check on `totalIdsAndAmounts` length.

```
if or(gt(totalIdsAndAmounts, 0xffffffff), iszero(eq(mload(0), shl(224,
↪  _AUTHORIZE_CLAIM_SELECTOR))))
```

The transaction would likely revert with `MemoryLimitOOG` first before reaching this line. Consider using a smaller maximum length bound, or removing the check.

4. `MetadataLib.readDecimalsWithDefaultValue()` is implemented but unused.

5. `HashLib.toBatchMessageHash()` is implemented but unused.
6. Checking `(assignableAt < block.timestamp).or(assignableAt > type(uint96).max)` in `EmissaryLib.sol:L146` is practically always false.

Uniswap Labs: Fixed in commit 5d453793.

Spearbit: Fix verified.

5.6.2 `toAllocatorIdIfRegistered()` doesn't revert for the null address

Severity: Informational

Context: IdLib.sol#L131-L143

Description: Passing the null address into `toAllocatorIdIfRegistered()` returns an allocator ID of `0xf000000000000000000000` even though the null address isn't registered, because the conditional check isn't entered for it. Thankfully, the current impact is negligible as this function is only used in `MockEmissaryLogic`. From observation, this function has been deprecated in favour of `hasRegisteredAllocatorId()`, which has a simpler and better check.

Proof of Concept:

```
function testToAllocatorIdIfRegistered_ToNullAddress() public {
   // does not revert even though null address is not registered
   // allocatorID := compact flag 0xf + nullAddress << 88
   assertEq(address(0).toAllocatorIdIfRegistered(), 0xf000000000000000000000000);
}
```

Recommendation: Revert if `allocator` is null.

Uniswap Labs: Fixed in commit 504a0ae3.

Spearbit: Fix verified.

5.6.3 Single native deposits allow 0 amount unlike its ERC20 counterparts

Severity: Informational

Context: DirectDepositLogic.sol#L156-L165

Description: For ERC20 deposits, zero deposit amounts revert with `InvalidDepositBalanceChange()`. However, for native deposits, only the batch version will revert for a zero amount with `InvalidBatchDepositStructure()`; single native deposits will not.

Proof of Concept:

```
function test_zeroValueEthDeposit() public {
    address recipient = swapper;
    ResetPeriod resetPeriod = ResetPeriod.TenMinutes;
    Scope scope = Scope.Multichain;
    uint256 amount = 0;

    vm.prank(allocator);
    uint96 allocatorId = theCompact.__registerAllocator(allocator, "");

    bytes12 lockTag =
        bytes12(bytes32((uint256(scope) << 255) | (uint256(resetPeriod) << 252) | (uint256(allocatorId)
        ↪  << 160)));

    vm.prank(swapper);
    uint256 id = theCompact.depositNative{ value: amount }(lockTag, address(0));
    assertEq(theCompact.balanceOf(recipient, id), amount);
}
```

Recommendation: Ensure `msg.value` is non-zero for single native deposits by adding the check in `_performCustomNativeTokenDeposit()`.

Uniswap Labs: Fixed in commit b1f2a1f8.

Spearbit: Fix verified.

5.6.4 Interest rebasing tokens will have interest locked in the contract

Severity: Informational

Context: TransferLib.sol#L141

Description: Resource locks keep track of the amount of tokens deposited, crediting that amount as the amount of ERC6909 tokens to mint to the depositor. Upon withdrawing those tokens, we burn the same amount of ERC6909 tokens as the amount of ERC20 tokens which are removed from the contract. Attempting to withdraw more ERC20 tokens than ERC6909 tokens held will result in a revert due to insufficient balance. As a result, interest rebasing tokens like stETH and Aave aTokens will have accrued interest locked in the contract.

Recommendation: Include documentation to indicate that rebasing tokens are unsupported.

Uniswap Labs: Fixed in commit 453ed949.

Spearbit: Fix verified.

5.6.5 Lack of signature replay protection may lead to unexpected effects

Severity: Informational

Context: RegistrationLogic.sol#L113-L138

Description: In `RegistrationLogic._deriveClaimHashAndRegisterCompact`, used by `*registerFor` functions, we produce a `claimHash` from the provided parameters and ensure that the sponsor signed for this `claimHash` before registering the compact:

```
function _deriveClaimHashAndRegisterCompact(
    address sponsor,
    bytes32 typehash,
    uint256 preimageLength,
    bytes32 domainSeparator,
    bytes calldata sponsorSignature
) internal returns (bytes32 claimHash) {
    assembly ("memory-safe") {
        // Retrieve the free memory pointer; memory will be left dirtied.
        let m := mload(0x40)

        // Copy relevant arguments from calldata to prepare hash preimage.
        // Note that provided arguments may have dirty upper bits, which will
        // give a claim hash that cannot be derived during claim processing.
        calldatacopy(m, 0x04, preimageLength)

        // Derive the claim hash from the prepared preimage data.
        claimHash := keccak256(m, preimageLength)
    }

    // Ensure that the sponsor has verified the supplied claim hash.
    claimHash.hasValidSponsor(sponsor, sponsorSignature, domainSeparator);

    // Register the compact for the indicated sponsor.
    sponsor.registerCompact(claimHash, typehash);
}
```

Notice that there is no signature replay protection implemented in this logic. Additionally, `registerCompact` does not check whether the compact has already been registered, instead it overwrites the registration timestamp. In the case of an ECDSA signature, it's impossible for the signer to revoke the signature.

In practice, the only consequence of this is that the registration period during which a new signature is not required to process a claim can be extended by replaying the signature to re-register the compact. This is not inherently concerning as the sponsor should have provided a reasonable `expires` timestamp beyond which the claim would be invalid regardless, but it may be unexpected behavior.

Recommendation: Consider either implementing signature replay protection or simply allowing the registration to remain valid until the provided `expires` timestamp.

Uniswap Labs: Fixed in commit c8347375.

Spearbit: Fix verified.

5.6.6 `burn()` doesn't revert for 0 amounts

Severity: Informational

Context: TransferLib.sol#L49-L50, TransferLib.sol#L246-L271

Description: For the 3 possible options for processing token operations, 2 of them don't revert for zero amounts. Only the 2nd case where `claimantLockTag == lockTag` will revert, when `release()` is called:

```
// Revert if amount is zero or exceeds balance.
if or(iszero(amount), gt(amount, fromBalance)) {
   mstore(0x00, 0xf4d678b8) // `InsufficientBalance()`.
   revert(0x1c, 0x04)
}
```

Proof of Concept:

```
function test_processTransfer_zeroAmountDirectWithdrawal() public {
  // Create components array with zero amount
  Component[] memory recipients = new Component[](1);
  recipients[0] = Component({ claimant: abi.decode(abi.encodePacked(bytes12(0), recipient), (uint256)),
  ↪  amount: 0 });

  // Create AllocatedTransfer struct
  AllocatedTransfer memory transfer = AllocatedTransfer({
      nonce: 1,
      expires: block.timestamp + 1 days,
      id: testTokenId,
      recipients: recipients,
      allocatorData: bytes("")
  });

  vm.prank(sponsor);
  // should revert, but doesn't
  logic.processTransfer(transfer);
}
```

Recommendation: Revert for zero amounts in `TransferLib.burn()` to be consistent with `release()`.

```
- if gt(amount, fromBalance) {
+ if or(iszero(amount), gt(amount, fromBalance)) {
     mstore(0x00, 0xf4d678b8) // `InsufficientBalance()`.
     revert(0x1c, 0x04)
  }
```

Uniswap Labs: Fixed in commit 4af07ce1.

Spearbit: Fix verified.

5.6.7 For-loop instead of nested-function for early return can improve readability

Severity: Informational

Context: DepositViaPermit2Lib.sol#L140-L148

Description: To achieve "early-return" semantics from within inline-assembly the referenced code defines a nested assembly-function within the top-level `writeWitnessAndGetTypehashes` function to be able to leverage the `leave` assembly operation. This however reduces readability as it requires renaming and aliasing the function's existing parameters.

Recommendation: Replace the inner function and `leave` with an assembly `for {} 1 {}` structure using `break` as the early exit. An additional `break` at the end is required to prevent an infinite loop:

```
      function writeWitnessAndGetTypehashes(
          uint256 memoryLocation,
          CompactCategory category,
 @@ -142,46 +161,47 @@ library DepositViaPermit2Lib {
              // Internal assembly function for writing the witness and typehashes.
              // Used to enable leaving the inline assembly scope early when the
              // witness is empty (no-witness case).
-             function writeWitnessAndGetTypehashes(memLocation, c, witnessOffset, witnessLength,
↪  usesBatch) ->
-                 derivedActivationTypehash,
-                 derivedCompactTypehash
-             {
+             for { } 1 { } {
                  // Derive memory offset for the witness typestring data.
-                 let memoryOffset := add(memLocation, 0x20)
+                 let memoryOffset := add(memoryLocation, 0x20)

                  // Declare variables for start of Activation and Category-specific data.
                  let activationStart
                  let categorySpecificStart

                  // Handle non-batch cases.
-                 if iszero(usesBatch) {
+                 if iszero(usingBatch) {
                      // Prepare initial Activation witness typestring fragment.
                      mstore(add(memoryOffset, 0x1b),
                      ↪  PERMIT2_DEPOSIT_WITH_ACTIVATION_TYPESTRING_FRAGMENT_TWO)
                      mstore(memoryOffset, PERMIT2_DEPOSIT_WITH_ACTIVATION_TYPESTRING_FRAGMENT_ONE)

                      // Set memory pointers for Activation and Category-specific data start.
                      activationStart := add(memoryOffset, 0x13)
                      categorySpecificStart := add(memoryOffset, 0x3b)
                  }

                  // Proceed with batch case if preparation of activation has not begun.
                  if iszero(activationStart) {
+                     // TODO: Review path
                      // Prepare initial BatchActivation witness typestring fragment.
                      mstore(memoryOffset,
                      ↪  PERMIT2_BATCH_DEPOSIT_WITH_ACTIVATION_TYPESTRING_FRAGMENT_ONE)
                      mstore(add(memoryOffset, 0x28),
                      ↪  PERMIT2_BATCH_DEPOSIT_WITH_ACTIVATION_TYPESTRING_FRAGMENT_THREE)
                      mstore(add(memoryOffset, 0x20),
                      ↪  PERMIT2_BATCH_DEPOSIT_WITH_ACTIVATION_TYPESTRING_FRAGMENT_TWO)

                      // Set memory pointers for Activation and Category-specific data.
```

```
                    activationStart := add(memoryOffset, 0x18)
                    categorySpecificStart := add(memoryOffset, 0x48)
                }

                // Declare variable for end of Category-specific data.
                let categorySpecificEnd

                // Handle Compact (non-batch, single-chain) case.
-               if iszero(c) {
+               if iszero(category) {
                    // Prepare next typestring fragment using Compact witness typestring.
                    mstore(categorySpecificStart, PERMIT2_ACTIVATION_COMPACT_TYPESTRING_FRAGMENT_ONE)
                    mstore(add(categorySpecificStart, 0x20),
                    ↪  PERMIT2_ACTIVATION_COMPACT_TYPESTRING_FRAGMENT_TWO)
                    mstore(add(categorySpecificStart, 0x40),
                    ↪  PERMIT2_ACTIVATION_COMPACT_TYPESTRING_FRAGMENT_THREE)
@@ -189,13 +209,14 @@ library DepositViaPermit2Lib {
                    mstore(add(categorySpecificStart, 0x60),
                    ↪  PERMIT2_ACTIVATION_COMPACT_TYPESTRING_FRAGMENT_FOUR)

                    // Set memory pointers for Activation and Category-specific data end.
                    categorySpecificEnd := add(categorySpecificStart, 0x88)
                    categorySpecificStart := add(categorySpecificStart, 0x10)
                }

                // Handle BatchCompact (single-chain) case.
-               if iszero(sub(c, 1)) {
+               if iszero(sub(category, 1)) {
                    // Prepare next typestring fragment using BatchCompact witness typestring.
                    mstore(categorySpecificStart,
                    ↪  PERMIT2_ACTIVATION_BATCH_COMPACT_TYPESTRING_FRAGMENT_ONE)
                    mstore(add(categorySpecificStart, 0x20),
                    ↪  PERMIT2_ACTIVATION_BATCH_COMPACT_TYPESTRING_FRAGMENT_TWO)
                    mstore(add(categorySpecificStart, 0x40),
                    ↪  PERMIT2_ACTIVATION_BATCH_COMPACT_TYPESTRING_FRAGMENT_THREE)
@@ -203,8 +224,8 @@ library DepositViaPermit2Lib {
                    mstore(add(categorySpecificStart, 0x60),
                    ↪  PERMIT2_ACTIVATION_BATCH_COMPACT_TYPESTRING_FRAGMENT_FOUR)

                    // Set memory pointers for Activation and Category-specific data end.
                    categorySpecificEnd := add(categorySpecificStart, 0x93)
                    categorySpecificStart := add(categorySpecificStart, 0x15)
                }

                // Revert on MultichainCompact case or above (registration only applies to the
                ↪  current chain).
@@ -215,35 +236,35 @@ library DepositViaPermit2Lib {
                }

                // Handle no-witness cases.
-               if iszero(witnessLength) {
+               if iszero(witness.length) {
                    // Derive memory offset for region used to retrieve typestring fragment by index.
-                   let indexWords := shl(5, c)
+                   let indexWords := shl(5, category)

                    // Prepare token permissions typestring fragment.
                    mstore(add(categorySpecificEnd, 0x0e), TOKEN_PERMISSIONS_TYPESTRING_FRAGMENT_TWO)
                    mstore(sub(categorySpecificEnd, 1), TOKEN_PERMISSIONS_TYPESTRING_FRAGMENT_ONE)

                    // Derive total length of typestring and store at start of memory.
-                   mstore(memLocation, sub(add(categorySpecificEnd, 0x2e), memoryOffset))
```

```
+                    mstore(memoryLocation, sub(add(categorySpecificEnd, 0x2e), memoryOffset))

                     // Derive activation typehash based on the compact category for non-batch cases.
-                    if iszero(usesBatch) {
+                    if iszero(usingBatch) {
                         // Prepare typehashes for Activation.
                         mstore(0, COMPACT_ACTIVATION_TYPEHASH)
                         mstore(0x20, BATCH_COMPACT_ACTIVATION_TYPEHASH)

                         // Retrieve respective typehash by index.
-                        derivedActivationTypehash := mload(indexWords)
+                        activationTypehash := mload(indexWords)
                     }

                     // Derive activation typehash for batch cases if typehash is not yet derived.
-                    if iszero(derivedActivationTypehash) {
+                    if iszero(activationTypehash) {
                         // Prepare typehashes for BatchActivation.
                         mstore(0, COMPACT_BATCH_ACTIVATION_TYPEHASH)
                         mstore(0x20, BATCH_COMPACT_BATCH_ACTIVATION_TYPEHASH)

                         // Retrieve respective typehash by index.
-                        derivedActivationTypehash := mload(indexWords)
+                        activationTypehash := mload(indexWords)
                     }

                     // Prepare compact typehashes.
  @@ -251,35 +272,32 @@ library DepositViaPermit2Lib {
                     mstore(0x20, BATCH_COMPACT_TYPEHASH)

                     // Retrieve respective typehash by index.
-                    derivedCompactTypehash := mload(indexWords)
+                    compactTypehash := mload(indexWords)

                     // Leave the inline assembly scope early.
-                    leave
+                    break
                 }

                 // Copy the supplied compact witness from calldata.
-                calldatacopy(categorySpecificEnd, witnessOffset, witnessLength)
+                calldatacopy(categorySpecificEnd, witness.offset, witness.length)

                 // Insert tokenPermissions typestring fragment.
-                let tokenPermissionsFragmentStart := add(categorySpecificEnd, witnessLength)
+                let tokenPermissionsFragmentStart := add(categorySpecificEnd, witness.length)
                 mstore(add(tokenPermissionsFragmentStart, 0x0f),
                 ↪  TOKEN_PERMISSIONS_TYPESTRING_FRAGMENT_TWO)
                 mstore(tokenPermissionsFragmentStart, TOKEN_PERMISSIONS_TYPESTRING_FRAGMENT_ONE)

                 // Derive total length of typestring and store at start of memory.
-                mstore(memLocation, sub(add(tokenPermissionsFragmentStart, 0x2f), memoryOffset))
+                mstore(memoryLocation, sub(add(tokenPermissionsFragmentStart, 0x2f), memoryOffset))

                 // Derive activation typehash.
-                derivedActivationTypehash :=
+                activationTypehash :=
                     keccak256(activationStart, sub(add(tokenPermissionsFragmentStart, 1),
                     ↪  activationStart))

                 // Derive compact typehash.
-                derivedCompactTypehash :=
```

```
+                   compactTypehash :=
+                       keccak256(categorySpecificStart, sub(add(tokenPermissionsFragmentStart, 1),
↪  categorySpecificStart))
+                   break
                }
-
-               // Execute internal assembly function and store derived typehashes.
-               activationTypehash, compactTypehash :=
-                   writeWitnessAndGetTypehashes(memoryLocation, category, witness.offset,
↪  witness.length, usingBatch)
           }
       }
```

Uniswap Labs: Fixed in commit 8f4b7937.

Spearbit: Fix verified.

5.6.8 Conflicting Storage Layout Convention may lead to unsuspecting collisions

Severity: Informational

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

Description: `TheCompact` follows the convention of using `(4-byte storage map selector) ...key bytes` as the pre-image for the storage slots of its custom mappings, e.g.:

```
function _getCutoffTimeSlot(address account, uint256 id) private pure returns (uint256
↪  cutoffTimeSlotLocation) {
    assembly ("memory-safe") {
        // Retrieve the current free memory pointer.
        let m := mload(0x40)

        // Pack data for computing storage slot.
        mstore(0x14, account)
        mstore(0, _FORCED_WITHDRAWAL_ACTIVATIONS_SCOPE)
        mstore(0x34, id)

        // Compute storage slot from packed data.
        cutoffTimeSlotLocation := keccak256(0x1c, 0x38)

        // Restore the free memory pointer.
        mstore(0x40, m)
    }
}
```

Where the inline assembly is a more efficient implementation of `cutoffTimeSlotLocation = bytes.concat(bytes4(_FORCED_WITHDRAWAL_ACTIVATIONS_SCOPE), bytes20(account), bytes32(id))`. This means that no matter the size of serialized key as long as unique leading selector bytes are used the 32-byte storage slots should never collide between different mappings and keys as long as `keccak256` remains sound.

The problem arises when utilizing libraries like Solady which have a different convention for deriving storage slots. (From Solady's code in `ERC6909.sol`):

```
let ownerSlotSeed := or(_ERC6909_MASTER_SLOT_SEED, shl(96, owner))
```

The balance slot of `owner` is given by:

```
mstore(0x20, ownerSlotSeed)
mstore(0x00, id)
let balanceSlot := keccak256(0x00, 0x40)
```

The operator approval slot of `owner` is given by:

```
mstore(0x20, ownerSlotSeed)
mstore(0x00, operator)
let operatorApprovalSlot := keccak256(0x0c, 0x34)
```

The allowance slot of (`owner`, `spender`, `id`) is given by:

```
mstore(0x34, ownerSlotSeed)
mstore(0x14, spender)
mstore(0x00, id)
let allowanceSlot := keccak256(0x00, 0x54)
```

Unlike the Compact's mappings solady puts the "storage map selector" roughly in the middle of the key bytes. This means that if the compact happens to define a mapping who's hash pre-image has the same length as one of Solady's mappings this would likely enable very dangerous collisions where-by e.g. a write to one given mapping with a weird address might derive a slot that's valid for another mapping.

While no such cases have been identified as the Compact doesn't seem to currently use `0x34`, `0x54` or `0x40` as lengths for its own map pre-images this is a relatively subtle fact that could cause a simple good faith modification in the future create a potentially critical issue.

Recommendation: Create contributor guidelines to ensure that people are aware of this fact and carefully review the storage slot derivation of new storage maps or any modifications made to existing maps. Furthermore a custom static analysis pass in a framework like wake added to the CI may help to more systemtically monitor for uses of hashes in storage layouts that may collide with libraries or other files.

Uniswap Labs: Fixed in commit 453ed949 .

Spearbit: Fix verified.

5.6.9 Use of Unallocated Free Pointer Across Scopes

Severity: Informational

Context: DepositViaPermit2Lib.sol#L67-L122, DepositViaPermit2Logic.sol#L454-L472

Description: To efficiently encode a dynamically sized call payload the memory free pointer is used *without* being allocated and passed across scopes to allow the data to be encoded in parts in different functions. This is efficient as it accommodates the dynamic size of the payload without incurring the cost of allocating & reallocating as more is added to the payload. This is quite brittle however because any accidental allocation along the way will cause the free pointer to change and the data being encoded to be corrupted.

Recommendation: Add some kind of testing to ensure that memory safety is not violated in the code. One potential approach is free pointer poisoning where while in use the free pointer is temporarily set to a very high memory address (e.g. via `mstore(0x40, 0xffffffffffff)`) such that it'd cause an Out-of-Gas (OOG) error if any other part of the code where to use it before the call encoding is complete. This would cause errors during testing letting you catch the error. Once the encoding is complete the free pointer can be restored to allow for safe use (e.g. via `mstore(0x40, m)`).

Uniswap Labs: Fixed in commit ff0107bc.

Spearbit: Fix verified.

5.6.10 Can Increase Readability by Reordering Struct Fields & Inlining Small Functions

Severity: Informational

Context: ClaimHashLib.sol#L67, ClaimHashLib.sol#L111-L118, BatchClaims.sol#L13-L22, BatchMultichainClaims.sol#L6-L30, Claims.sol#L14-L25, MultichainClaims.sol#L6-L34

Description: To fit in the 24kB spurious dragon code size limit the Compact employs a variety of tricks to be able to reuse functions between different but similar struct types, specifically it uses raw calldata pointers and

relative calldata offsets in the shape of `uint256` values as its primary type for different functions. However several simplifications can be made to improve clarity *without* sacrificing code size and in some cases improving it:

1. Inlining `_toGenericMultichainClaimWithWitnessMessageHash` in `src/lib/ClaimHashLib.sol`.
2. Reorder the fields of `MultichainClaim` and `ExogenousMultichainClaim` to match the order of the fields in `Claim` such that `_processClaim`, `_processMultichainClaim` and `_processExogenousMultichainClaim` can all directly call `processClaimWithComponents` without the current indirection through `processSimpleClaim` and `processClaimWithSponsorDomain`.
3. Similarly the fields in `BatchMultichainClaim` and `ExogenousBatchMultichainClaim` can be reordered to match `BatchClaim` so that `_processBatchClaim`, `_processBatchMultichainClaim` and `_processExogenousBatchMultichainClaim` can all directly call `processClaimWithBatchComponents` without indirection.
4. Remove function cast on call to `HashLib.toMessageHashWithWitness` in `ClaimHashLib.toMessageHashes(Claim calldata claim)` as it's the only use of `toMessageHashWithWitness`.

Recommendation: Apply suggested simplifications to improve readability.

Uniswap Labs: Fixed in PR 142.

Spearbit: Fix verified.