hashlock.



Security Audit

InstaSwap 3rd (DeFi)

November 2026 info@hashlock.com.au Hashlock Pty Ltd

Table of Contents

Executive Summary	4
Project Context	4
Audit Scope	7
Security Rating	8
Intended Smart Contract Functions	9
Code Quality	12
Audit Resources	12
Dependencies	12
Severity Definitions	13
Status Definitions	14
Audit Findings	15
Centralisation	34
Conclusion	35
Our Methodology	36
Disclaimers	38
About Hashlock	39

CAUTION

THIS DOCUMENT IS A SECURITY AUDIT REPORT AND MAY CONTAIN CONFIDENTIAL INFORMATION. THIS INCLUDES IDENTIFIED VULNERABILITIES AND MALICIOUS CODE WHICH COULD BE USED TO COMPROMISE THE PROJECT. THIS DOCUMENT SHOULD ONLY BE FOR INTERNAL USE UNTIL ISSUES ARE RESOLVED. ONCE VULNERABILITIES ARE REMEDIATED, THIS REPORT CAN BE MADE PUBLIC. THE CONTENT OF THIS REPORT IS OWNED BY HASHLOCK PTY LTD FOR USE OF THE CLIENT.

Executive Summary

The InstaSwap team partnered with Hashlock to conduct a security audit of their smart contracts. Hashlock manually and proactively reviewed the code in order to ensure the project's team and community that the deployed contracts are secure.

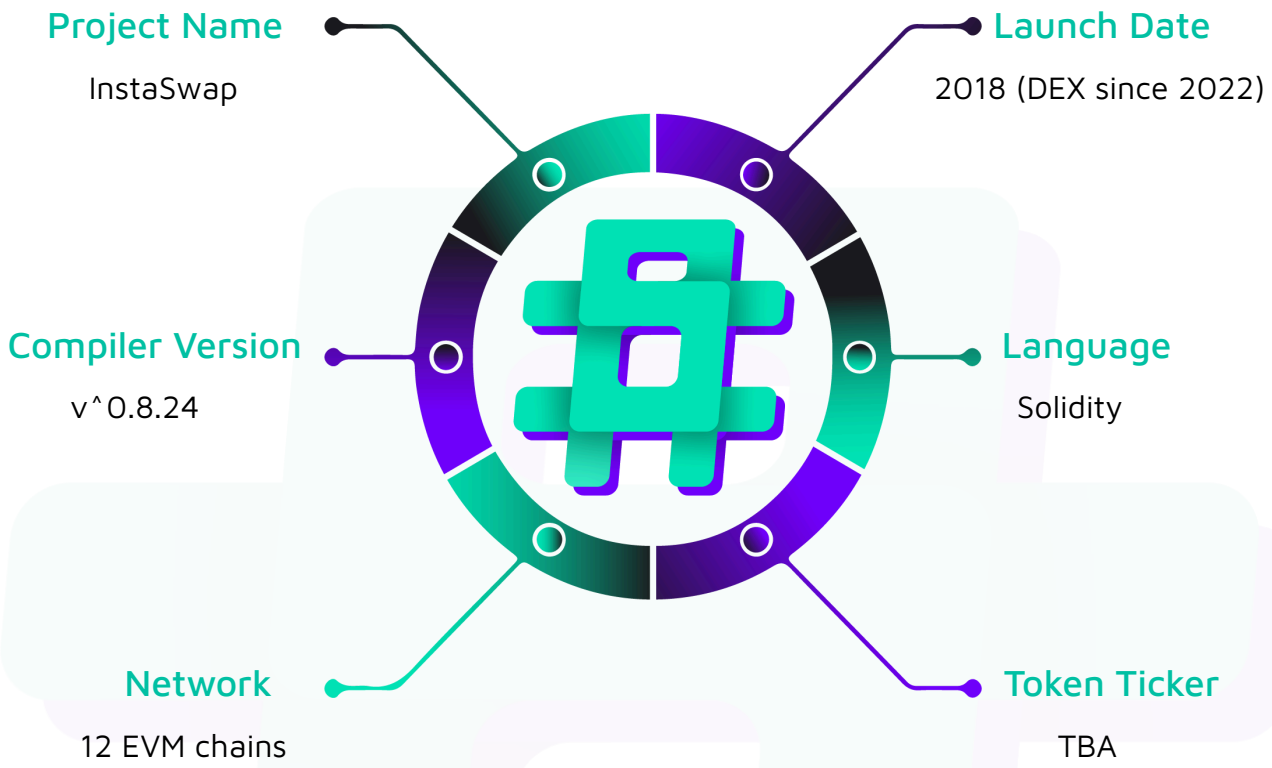
Project Context

InstaSwap is a non-custodial, cross-chain decentralized exchange (DEX) aggregator that enables users to swap native cryptocurrencies across 67+ blockchains by aggregating liquidity from multiple decentralized protocols. The platform also offers spot and perpetual trading, private payments, Split Swaps, and tokenized stocks, with a focus on cross-chain interoperability, self-custody, and privacy.

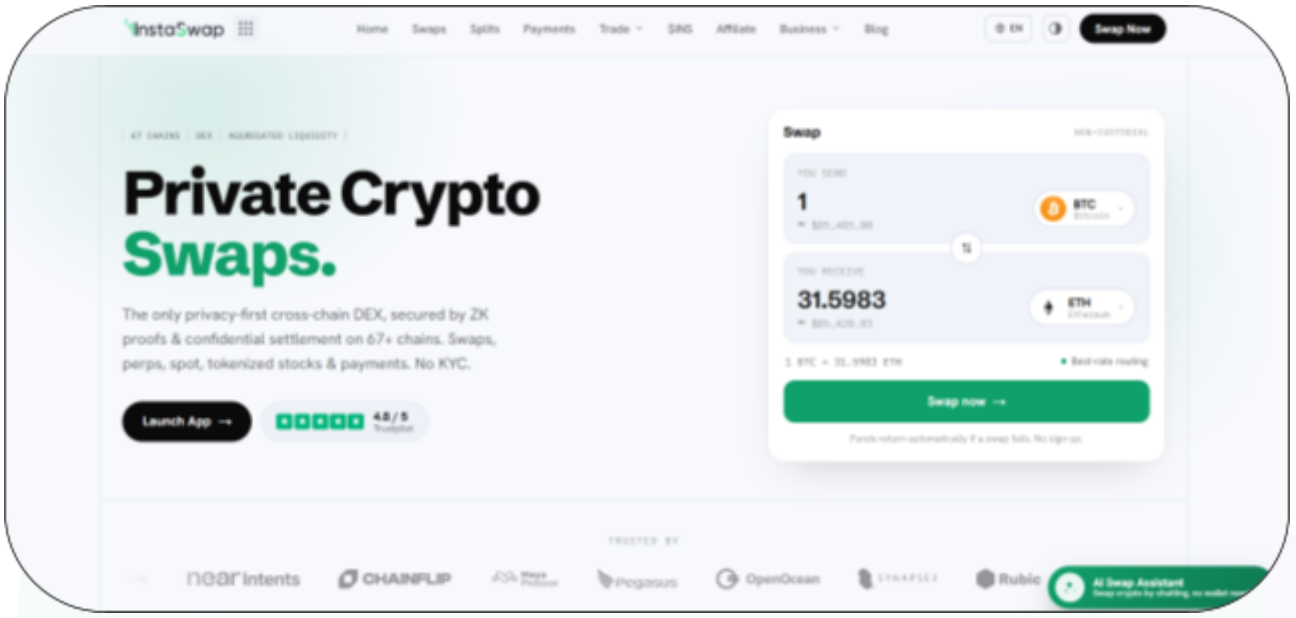
Project Name: InstaSwap
Project Type: Defi, DEX
Compiler Version: ^0.8.24
Website: https://instaswap.com/
Logo:

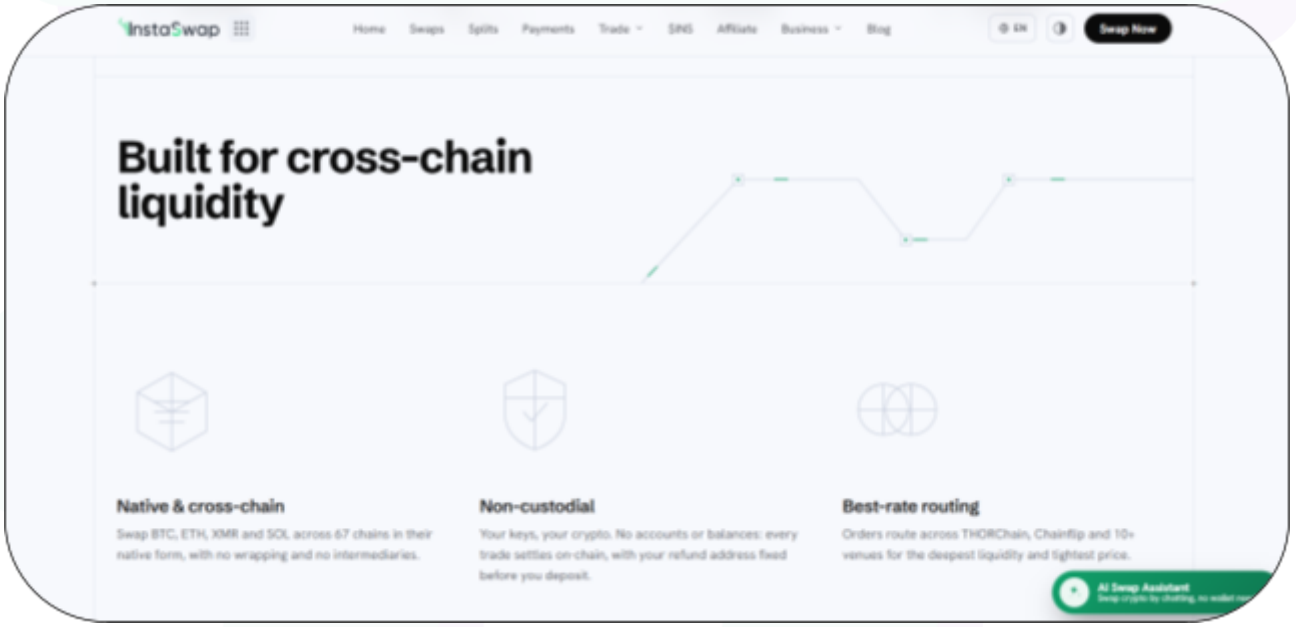


Visualised Context:



Project Visuals:





Audit Scope

We at Hashlock audited the solidity code within the InstaSwap project, the scope of work included a comprehensive review of the smart contracts listed below. We tested the smart contracts to check for their security and efficiency. These tests were undertaken primarily through manual line-by-line analysis and were supported by software-assisted testing.

Description	**InstaSwap Smart Contracts**
Network	**12 EVM chains**
Language	**Solidity**
Audit Date	**November, 2026**
Contract 1	FactoryGovernance
Contract 2	ApproverRegistryV2
Contract 3	EscrowFactoryV7
Contract 4	SplitEscrowFactoryV4
Contract 5	UniswapEscrowFactoryV4
Contract 6	UniswapV4EscrowFactoryV4
Contract 7	UniswapSwapEscrowV4
Contract 8	UniswapV4SwapEscrowV4
Audited GitHub Commit Hash	fbd3ba41f65049c7b923b81fec60327f6fa30df1
Fix Review GitHub Commit Hash	95739f73c9589a940acaf3c1df6e9a54063fd5d1

Note: *Hashlock initially audited the code associated with the "Audited GitHub Commit Hash" and the findings presented in this report pertain specifically to that commit. For the "Fix Review GitHub Commit Hash", Hashlock solely verified the status of the identified findings and did not conduct a comprehensive review to assess any additional code implementations.*

Security Rating

After Hashlock's Audit, we found the smart contracts to be **"Hashlocked"**. The contracts all follow simple logic, with correct and detailed ordering. They use a series of interfaces.



The 'Hashlocked' rating is reserved for projects that complete at least two security engagements with Hashlock, such as a second audit or a penetration test.

All issues uncovered during automated and manual analysis were meticulously reviewed and applicable vulnerabilities are presented in the Audit Findings section. The list of audited assets is presented in the Audit Scope section and the project's contract functionality is presented in the Intended Smart Contract Functions section.

All vulnerabilities initially identified have now been resolved.

Hashlock found:

1 Medium severity vulnerability

4 Low severity vulnerabilities

11 QA

Caution: *Hashlock's audits do not guarantee a project's success or ethics, and are not liable or responsible for security. Always conduct independent research about any project before interacting.*

Intended Smart Contract Functions

Claimed Behaviour	Actual Behaviour
FactoryGovernance.sol - Lets the owner make tightening changes at once: raise the approval threshold, remove an approver, pause. - Makes the owner schedule loosening changes (approver set, operator, guardian, ownership) with a 48-hour delay and a 7-day execution window. - Lets the owner or the guardian cancel any scheduled change or pause the factory (at the audited commit; in the fix review build, only the owner can cancel a guardian change, see Q-09). - Lets anyone execute a scheduled change once its delay has passed. - Holds the approval threshold above an immutable floor that no path can lower. - Keeps the treasury address immutable after deployment. - Does not let one address hold two roles (owner, guardian, operator, approver). - Lets ownership pass only by a two-step transfer.	**Contract achieves this functionality.**
ApproverRegistryV2.sol - Checks that a set number of registered approvers have signed each fund-moving action (EIP-712, version "10" at the audited commit and version "11" in the fix review build). - Invalidates old approvals when the approver set changes (approver epoch).	**Contract achieves this functionality.**

- Gives no path that skips the approval check. -	
EscrowFactoryV7.sol - Lets anyone deploy a swap escrow at a predictable CREATE2 deposit address for a given refund address (at the audited commit; in the fix review build, escrow creation requires the approvers' signatures, see M-01). - Lets the operator forward tokens or ETH to a destination after the approvers sign, with a signed cap on the fee. - Lets the operator run the second leg of a swap or bridge after the approvers sign. - Sends the leftover balance (up to maxFee) to the treasury and keeps the rest for the user. - Lets anyone refund the escrow balance to the user's refund address after the refund timeout. - Lets the operator sweep stray funds after the approvers sign, but not funds held for a deferred refund.	**Contract achieves this functionality.**
SplitEscrowFactoryV4.sol - Lets anyone deploy a split escrow at a predictable CREATE2 deposit address (at the audited commit; in the fix review build, escrow creation requires the approvers' signatures, see M-01). - Lets the operator forward one or several token or ETH amounts to destinations after the approvers sign. - Lets anyone refund the escrow balance to the user's refund address after the refund timeout.	**Contract achieves this functionality.**

- Lets the operator sweep stray funds after the approvers sign.	
UniswapEscrowFactoryV4.sol - Lets anyone deploy a Uniswap V3 swap escrow with a fixed refund address, recipient, input token and output token (at the audited commit; in the fix review build, escrow creation requires the approvers' signatures, see M-01). - Lets the operator swap the deposit through the Uniswap router and send the output to the recipient, after the approvers sign. - Lets the operator swap and hold the output in the escrow (swapHold) for later delivery. - Wraps and unwraps WETH for native ETH swaps. - Lets the operator forward ETH after the approvers sign, with a signed cap on the fee. - Lets anyone refund the escrow balance to the user's refund address after the refund timeout. - Lets the operator sweep stray funds after the approvers sign, but not held or deferred-refund funds.	**Contract achieves this functionality.**
UniswapV4EscrowFactoryV4.sol - Lets anyone deploy a Uniswap V4 swap escrow with a fixed refund address, recipient, input token and output token (at the audited commit; in the fix review build, escrow creation requires the approvers' signatures, see M-01). - Lets the operator swap through the V4 PoolManager (unlock callback) and send the output to the recipient, after the approvers sign. - Lets the operator swap and hold the output in	**Contract achieves this functionality.**

the escrow for later delivery. - Supports native ETH as the input or output token. - Lets the operator forward ETH after the approvers sign, with a signed cap on the fee. - Lets anyone refund the escrow balance to the user's refund address after the refund timeout. - Lets the operator sweep stray funds after the approvers sign, but not held or deferred-refund funds.	

Code Quality

This audit scope involves the smart contracts of the InstaSwap project, as outlined in the Audit Scope section. All contracts, libraries, and interfaces mostly follow standard best practices and to help avoid unnecessary complexity that increases the likelihood of exploitation, however, some refactoring was recommended to optimize security measures.

The code is very well commented on and closely follows best practice nat-spec styling. All comments are correctly aligned with code functionality.

Audit Resources

We were given the InstaSwap project smart contract code in the form of GitHub access.

As mentioned above, code parts are well commented. The logic is straightforward, and therefore it is easy to quickly comprehend the programming flow as well as the complex code logic. The comments are helpful in providing an understanding of the protocol's overall architecture.

Dependencies

As per our observation, the libraries used in this smart contracts infrastructure are based on well-known industry standard open source projects.
Apart from libraries, its functions are used in external smart contract calls.

Severity Definitions

The severity levels assigned to findings represent a comprehensive evaluation of both their potential impact and the likelihood of occurrence within the system. These categorizations are established based on Hashlock's professional standards and expertise, incorporating both industry best practices and our discretion as security auditors. This ensures a tailored assessment that reflects the specific context and risk profile of each finding.

Significance	Description
High	High-severity vulnerabilities can result in loss of funds, asset loss, access denial, and other critical issues that will result in the direct loss of funds and control by the owners and community.
Medium	Medium-severity issues should be resolved before deployment. While they do not typically lead to a complete loss of funds, they may result in partial loss of funds or unintended behavior under certain conditions.
Low	Low-level vulnerabilities are areas that lack best practices that may cause small complications in the future.
Gas	Gas Optimisations, issues, and inefficiencies.
QA	Quality Assurance (QA) findings are informational and don't impact functionality. Supports clients improve the clarity, maintainability, or overall structure of the code.

Status Definitions

Each identified security finding is assigned a status that reflects its current stage of remediation or acknowledgment. The status provides clarity on the handling of the issue and ensures transparency in the auditing process. The statuses are as follows:

Significance	Description
Resolved	The identified vulnerability has been fully mitigated either through the implementation of the recommended solution proposed by Hashlock or through an alternative client-provided solution that demonstrably addresses the issue.
Acknowledged	The client has formally recognized the vulnerability but has chosen not to address it due to the high cost or complexity of remediation. This status is acceptable for medium and low-severity findings after internal review and agreement. However, all high-severity findings must be resolved without exception.
Unresolved	The finding remains neither remediated nor formally acknowledged by the client, leaving the vulnerability unaddressed.

Audit Findings

Medium

[M-01] EscrowFactoryV7#createEscrow - Operator Can Deploy A Poisoned Deposit Address And Take The Deposit Without Approvals

Description

A customer deposits to an address that the order service quotes as `predictEscrow(salt, userRefund)`. In gen-9, `createEscrow()` needed 3 approver signatures over `(predicted, userRefund)`. Before they signed, the approvers rebuilt the address from their own view of the order. The `ApproverRegistry` NatSpec states the purpose: "a poisoned deposit address (attacker-chosen userRefund / recipient baked into the escrow) is never deployed and its unguarded refund path never exists".

Gen-10 deletes this check in all four factories. `createEscrow()` now takes no approvals, and no later guarded call binds `userRefund`. The refund path needs no signature: the operator can call `refundToUser()` at any time, and anyone can call it after `refundTimeout`. A party that controls the quote, such as a compromised operator backend, can now get a deposit out with zero approver signatures. The approval gate exists to stop exactly that party.

Vulnerability Details

The gen-9 version (removed):

```
_checkApprovals(predicted, this.createEscrow.selector, address(0), userRefund, 0,
bytes32(0), 0, deadline, approvals);
```

The gen-10 version:

```
function createEscrow(bytes32 salt, address userRefund) external returns (address
escrow) {
    _validateRefund(userRefund);          // zero, sentinel, factory, implementation
```

```
only
    ...
    SwapEscrowV7(payable(escrow)).initialize(userRefund);
}

function refundToUser(address token) external nonReentrant {
    if (msg.sender != _operator() && block.timestamp < _refundUnlockAt) revert
RefundLocked();
    ...
    _safeTransfer(token, userRefund, bal);   // no approval
}
```

The same pattern is in `SplitEscrowFactoryV4.createEscrow()`, `UniswapEscrowFactoryV4.createEscrow()` and `UniswapV4EscrowFactoryV4.createEscrow()`. In the two Uniswap factories the attacker also picks `recipient`. In the swap escrows, the remainder `back` of `_settleRemainder()` also goes to the poisoned `userRefund`.

Proof of Concept

1. The quote service is compromised. It gives Alice the address `predictEscrow(salt, Mallory)` instead of `predictEscrow(salt, Alice)`.
2. Alice deposits 100,000 USDC to that address.
3. The operator key calls `createEscrow(salt, Mallory)`. No approvals are checked.
4. In the same transaction, the operator calls `refundToUser(USDC)`. The operator does not wait for the unlock, so 100,000 USDC goes to Mallory.
5. Without the operator key, Mallory does the same after `refundTimeout` (24 h).

In gen-9, step 3 reverts with `ApprovalsInsufficient`. The deposit then stays at an address without code, and no one can take it.

```
address payable quoted = payable(factory.predictEscrow(SALT, MALLORY));
usdc.mint(quoted, 100_000e6);                          // Alice deposits
vm.startPrank(OPERATOR);
factory.createEscrow(SALT, MALLORY);                  // no approvals
SwapEscrowV7(quoted).refundToUser(address(usdc));      // operator skips the unlock
vm.stopPrank();
```

```
assertEq(usdc.balanceOf(MALLORY), 100_000e6);          // whole deposit taken, 0
signatures
```

Impact

A compromised operator, or anyone who controls the quoted parameters, takes the full deposit of every customer who received a poisoned address, and no approver signs anything. The 3-of-5 approver floor, which gen-10 hardens elsewhere, no longer protects this path. That breaks the stated invariant that the operator moves funds only with approvals. Likelihood is low because it needs an off-chain compromise. Impact is high because the whole deposit is lost.

Recommendation

It is recommended to keep an approval subject on `createEscrow()`, so the approvers still check `userRefund` (and `recipient`) against the customer's order before the refund path can exist. The public recovery endpoint can keep serving third-party deployment as it did in gen-9.

```
-    function createEscrow(bytes32 salt, address userRefund) external returns
(address escrow) {
+    function createEscrow(bytes32 salt, address userRefund, uint256 deadline,
bytes[] calldata approvals)
+        external returns (address escrow)
+    {
         _validateRefund(userRefund);
         bytes32 s = _salt(salt, userRefund);
         address predicted = Clones1167.predict(implementation, s, address(this));
+        _checkApprovals(predicted, this.createEscrow.selector, address(0),
userRefund, 0, bytes32(0), 0, deadline, approvals);
         escrow = Clones1167.clone(implementation, s);
```

Status

Resolved

Low

[L-01] EscrowFactoryV7#createEscrow - Anyone Can Start The Refund Timer Before The Deposit Arrives

Description

`initialize()` sets `_refundUnlockAt` from the block where the clone is deployed. In gen-10 anyone can call `createEscrow()`, and the CREATE2 address does not commit to the unlock time. A caller who knows the quote parameters can deploy early, so the deposit arrives already refundable, and any caller can refund it before the operator acts. This breaks the client's invariant that the address commits to every value written into the clone. All four factories are affected.

Recommendation

Commit a quote expiry to the CREATE2 salt and revert after it, or start the timer at the first guarded call.

Status

Resolved

[L-02] UniswapV4SwapEscrowV4#sweep - Deferred Refund Has No Exit When The Refund Address Cannot Receive It

Description

Gen-10 removes `deferredRefund` from `sweep()` / `sweepEth()`, so the only exit is `refundToUser()` / `refundEthToUser()`. Both send to the same `userRefund` that already failed. If a token blocklists `userRefund`, or `userRefund` rejects ETH (for example, the leftover held ETH after `forwardEth()`), the amount stays in the escrow forever. The v4 escrow is one-shot, so no second swap can move it. In gen-9, an approved sweep could recover it. `UniswapSwapEscrowV4` has the same gap; its only other exit is a second swap (Q-01).

Recommendation

Add an approval-gated function that sends `deferredRefund[token]` to a signed address and clears the record.

Status

Resolved

[L-03] FactoryGovernance#cancelApprovers - Guardian Safe Has The Same Signers As The Owner Safe

Description

The guardian is the only brake on a captured owner during the 48 h delay. In the deployment records, the owner Safe and the guardian Safe are both 2-of-3 with the same three signers (`0x7815…65D7`, `0x05c0…07Ac`, `0x4e10…Ac4A`). Any two signers can schedule their own approver set (which passes the floor of 3) and a new operator, and the same quorum is the only party that could cancel it. So the guardian adds no independent check.

Recommendation

Give the guardian Safe signers that do not overlap with the owner Safe, and record this in the centralisation section.

Status

Resolved

[L-04] UniswapV4SwapEscrowV4#sweepEth - Operator Can Sweep A Deferred USDC Refund On Arc As Native Value

Description

When the customer leg of a settlement fails, for example because USDC blocklists `userRefund`, the escrow keeps the amount as `deferredRefund[token]`. The contract treats that amount as the customer's: `sweep()` and `sweepEth()` must never take it.

On Arc (chain 5042), USDC is the native gas asset (18 decimals) and also the ERC-20 at `0x3600000000000000000000000000000000000000` (6 decimals). Both forms share one balance. The escrow tracks the two forms under different keys:

- a failed ERC-20 leg writes `deferredRefund[0x3600…]`;
- `_sweepableNative()` subtracts only `heldAmount + deferredRefund[address(0)]`.

```
function _sweepableNative() internal view returns (uint256) {
    uint256 reserved = heldAmount + deferredRefund[address(0)];
    uint256 bal = address(this).balance;
    return bal > reserved ? bal - reserved : 0;
}
```

On Arc with `nativeAllowed == false`, nothing ever writes `deferredRefund[address(0)]` or `heldAmount`. So `sweepEth()` sees the whole USDC balance as sweepable, including the customer's deferred ERC-20 refund. After the native sweep, the ERC-20 balance is lower than `deferredRefund[0x3600…]`, and `redirectDeferred(0x3600…)` reverts.

The same split affects `refundEthToUser()`. It sends the whole balance, but deletes only `deferredRefund[address(0)]`, so a stale `deferredRefund[0x3600…]` remains with no balance under it.

The repository targets this case: `chains.manifest.js` configures Arc with `nativeEnabled: false` and notes that USDC is both the native gas asset and an ERC-20. Every `UniswapV4EscrowFactoryV4` built from this code for Arc is affected.

The issue was introduced in gen-10, when the deferred-refund reserve was added.

Gen-9 had no reserve, and its `sweep()` / `sweepEth()` took the whole balance by design. Gen-11 carries the gen-10 logic unchanged. It only adds the signed `amount`, capped by the same `_sweepableNative()`.

Impact

The operator, with three approver signatures for a `sweepEth()` on an Arc escrow, sends a customer's deferred USDC refund to the treasury. This breaks the on-chain guarantee that a deferred refund is never sweepable. The record then points at a balance that no longer exists, so `redirectDeferred()` can no longer pay the customer. The path needs the operator and the approvers, and only escrows on Arc are affected.

Recommendation

It is recommended to stop the native paths on a factory that does not allow native value, so that one balance cannot be tracked under two keys. Until a new generation is deployed, the signer policies should refuse every `sweepEth()` on Arc.

```
function sweepEth(uint256 amount, uint256 deadline, bytes[] calldata approvals)
external onlyOperator nonReentrant {
+         if (!nativeAllowed) revert NativeNotEnabled();
          address t = IUniswapV4EscrowFactoryV4(factory).treasury();
          if (amount == 0) revert ZeroAmount();
          if (amount > _sweepableNative()) revert AmountExceedsSweepable();
```

```
function refundEthToUser() external nonReentrant {
+         if (!nativeAllowed) revert NativeNotEnabled();
          if (userRefund == address(0)) revert ZeroAddress();
```

Status

Resolved

QA

[Q-01] UniswapSwapEscrowV4#_pullIn - A Second Swap Can Spend The Customer's Deferred Refund

Description

When a best-effort refund fails, the escrow records the amount in `deferredRefund` and keeps it out of fees and sweeps. `_pullIn()` checks `amountIn` against the raw balance, not against `_sweepable()`, and the router escrow has no one-shot latch. So a second approved `swapForward()` or `swapHold()` can spend the customer's deferred refund, and the record then stays above the real balance. `SwapEscrowV7.forward()`, `forwardNative()`, `executeLeg2()` and `executeLeg2Native()` have the same gap; that code is unchanged since gen-9.

Recommendation

It is recommended to bound the swap input by the free balance, so a deferred refund can never be used as input.

```
function _pullIn(address w, uint256 amountIn) internal {
         address tIn = tokenIn;
         if (tIn == address(0)) {
-            if (address(this).balance < amountIn) revert InsufficientDeposit();
+            if (_sweepableNative() < amountIn) revert InsufficientDeposit();
             IWETH9V4(w).deposit{ value: amountIn }();
-        } else if (_balanceOf(tIn) < amountIn) {
+        } else if (_sweepable(tIn, _balanceOf(tIn)) < amountIn) {
             revert InsufficientDeposit();
         }
     }
```

Status

Resolved

[Q-02] FactoryGovernance#constructor - Immutable Treasury Cannot Be Replaced

Description

In gen-10, `treasury` is immutable, and every fee leg uses `_safeTransfer()` / `_sendEth()`, which must succeed. If a token blocklists the treasury, or the treasury rejects ETH, every settlement with a fee above zero reverts for that asset. If the treasury key leaks, every later fee goes to the attacker. The only responses are to sign `maxFee = 0` or to deploy a new factory generation.

Recommendation

It is recommended to document this as an accepted residual, or to make the fee leg best effort, so a failed fee transfer is recorded instead of reverting the settlement.

```
-        if (fee > 0) { _safeTransfer(tIn, t, fee); emit Swept(tIn, t, fee); }
+        if (fee > 0) {
+            if (_tryTransfer(tIn, t, fee)) emit Swept(tIn, t, fee);
+            else { deferredFee[tIn] += fee; emit FeeDeferred(tIn, fee); }
+        }
```

Status

Resolved

[Q-03] FactoryGovernance#transferOwnership - Nominating A Pending Approver Jams executeApprovers And Escrow Creation

Description

`transferOwnership()` rejects a nominee that is a current approver, but not one in the pending approver set. While the nomination is pending, `executeApprovers()` reverts with `ApproverIsOwner`. The due change still counts in `looseningEffectiveAt()`, so `createEscrow()` and `swapHold()` revert with `LooseningImminent` until the 7-day window ends or someone cancels the change.

Recommendation

It is recommended to reject a nominee that is in the pending approver set.

```
-        if (isApprover[_pendingOwner]) revert ApproverIsOwner();
+        if (isApprover[_pendingOwner] || _inPendingApprovers(_pendingOwner)) revert
ApproverIsOwner();
```

Status

Resolved

[Q-04] UniswapSwapEscrowV4#forwardEth - Settlement Uses Native Balance The Approval Does Not Name

Description

`forwardEth()` signs `(to, amount, maxFee)`. After the push, it settles the whole `_sweepableNative()`: the held rest plus any stray native, such as ETH drained from the router. `maxFee` is capped at `amount`, not at the held rest, so the approvers never see the base they settle. The treasury never receives more than the signed `maxFee`. `UniswapV4SwapEscrowV4.forwardEth()` has the same code.

Recommendation

It is recommended to cap `maxFee` at the held rest, so the approvers sign a cap on the exact amount being settled.

```
if (amount > held) revert InsufficientDeposit();
-         if (maxFee > amount) revert FeeTooLarge();
+         if (maxFee > held - amount) revert FeeTooLarge();
```

Status

Resolved

[Q-05] UniswapSwapEscrowV4#forwardEth - Forward To The Escrow Itself Turns Held Output Into Fee

Description

`initialize()` rejects the escrow as `recipient`, but `forwardEth()` checks only `to != address(0)`. With `to = address(this)`, the ETH comes back through `receive()` after `heldAmount = 0`, and `_settleNative()` then pays up to `maxFee` of it to the treasury and the rest to the user. Nothing reaches the bridge. This needs the approvers to sign that target. `UniswapV4SwapEscrowV4.forwardEth()` has the same gap.

Recommendation

It is recommended to reject the escrow itself as the forward target.

```
-       if (to == address(0)) revert ZeroAddress();
+       if (to == address(0) || to == address(this)) revert ZeroAddress();
```

Status

Resolved

[Q-06] UniswapV4SwapEscrowV4#forwardEth - Partial Forwards Of Held Output No Longer Work

Description

Gen-10 sets `heldAmount = 0` on the first `forwardEth()` and settles the rest at once. A second forward of the same held output now reverts with `NothingHeld`. In the router escrow, two `swapHold()` calls add into one `heldAmount`, so a second bridge leg cannot run either. The contract notice still says the selectors are unchanged, but the `forwardEth` selector and data hash changed.

Recommendation

It is recommended to confirm that the operator never forwards a held output in parts and to correct the notice. If partial forwards are still needed, settle the rest only when the full held amount is forwarded.

```
-         heldAmount = 0;
+         heldAmount = held - amount;
          _sendEth(to, amount);
          emit EthForwarded(to, amount);
-         _settleNative(_sweepableNative(), maxFee);
+         if (heldAmount == 0) _settleNative(_sweepableNative(), maxFee);
```

Status

Resolved

[Q-07] UniswapSwapEscrowV4#swapHold - Repeated Holds Reset The Refund Timer, And A Zero Output Is Accepted

Description

Each `swapHold()` sets `_refundUnlockAt` again. The router escrow has no one-shot latch, so each further approved hold delays the customer's permissionless refund of ETH that was already held. With `minOut = 0`, a hold that returns zero output still uses an approval and still resets the timer, while `heldAmount` does not change.

Recommendation

It is recommended to reject a zero output and to restart the timer only for the first hold.

```
amountOut = _swap(effIn, w, uniPath, amountIn, minOut);
+        if (amountOut == 0) revert TooLittleOut();
         IWETH9V4(w).withdraw(amountOut);
+        bool firstHold = heldAmount == 0;
         heldAmount += amountOut;
-        _refundUnlockAt =
uint64(IUniswapEscrowFactoryV4(factory).creationUnlockAt());
+        if (firstHold) _refundUnlockAt =
uint64(IUniswapEscrowFactoryV4(factory).creationUnlockAt());
```

Status

Resolved

[Q-08] FactoryGovernance#_checkGuardianRole - Guardian Can Be An Approver Or The Treasury

Description

`_checkGuardianRole()` rejects only the owner and the operator, current and pending. `_checkApproverRole()` does not reject the guardian. So the brake and an approval signer can share one key. A guardian that equals the immutable treasury can be undone only by a new factory.

Recommendation

It is recommended to keep the guardian apart from the approvers and the treasury, in both directions.

```
function _checkGuardianRole(address g) internal view {
         if (g == address(0)) revert ZeroAddress();
         if (g == owner || g == pendingOwner || g == operator || g ==
pendingOperator.operator) revert GuardianConflict();
+        if (g == treasury || isApprover[g] || _inPendingApprovers(g)) revert
GuardianConflict();
     }
```

```
function _checkApproverRole(address a) internal view {
         ...
         if (a == treasury) revert TreasuryIsApprover();
+        if (a == guardian || a == pendingGuardian.guardian) revert
GuardianConflict();
     }
```

Status

Resolved

[Q-09] FactoryGovernance#cancelGuardian - Guardian Can Block Its Own Replacement

Description

`cancelGuardian()`, `cancelOwnershipTransfer()` and `pause()` are owner-or-guardian, while `unpause()` is owner-only. A leaked guardian key can cancel every attempt to replace it, and it can pause again after each unpause, until a new factory is deployed. Refunds still work. This code is unchanged since gen-9.

Recommendation

It is recommended to let only the owner cancel a guardian change, so the guardian cannot block its own removal.

```
-    function cancelGuardian() external onlyOwnerOrGuardian {
+    function cancelGuardian() external onlyOwner {
```

Status

Resolved

[Q-10] EscrowFactoryV7#createEscrow - Anyone Can Front-Run The Operator's createEscrow

Description

`createEscrow()` is open to everyone, and `Clones1167.clone()` reverts when the address already has code. A caller who copies the operator's pending call deploys the same escrow first, and the operator's transaction reverts. No funds are lost, but an operator transaction that bundles creation with another call fails. All four factories are affected.

Recommendation

It is recommended to return the existing escrow instead of reverting when the address is already deployed.

```
address predicted = Clones1167.predict(implementation, s, address(this));
+        if (predicted.code.length != 0) return predicted;
         escrow = Clones1167.clone(implementation, s);
```

Status

Resolved

[Q-11] SplitEscrowV4#sweep - One Wei Sent Before A Sweep Breaks The Signed Amount

Description

`sweep()` and `sweepEth()` sign the live balance as the amount. Any sender can add 1 wei before the call, and the digest then no longer matches the approvals. `SplitEscrowV4` has no inline fee settlement, so the refund user can repeat this until the unlock and then take the unswept fee with `refundToUser()`. All four escrows have the same shape. This code is unchanged since gen-9.

Recommendation

It is recommended to pass the swept amount as an argument and transfer exactly that amount, so a donation cannot change the signed value.

```
-    function sweep(address token, uint256 deadline, bytes[] calldata approvals)
external onlyOperator nonReentrant {
+    function sweep(address token, uint256 amount, uint256 deadline, bytes[] calldata
approvals) external onlyOperator nonReentrant {
         address t = ISplitEscrowFactoryV4(factory).treasury();
-        uint256 bal = _balanceOf(token);
-        if (bal == 0) revert ZeroAmount();
-        _approved(this.sweep.selector, token, t, bal, bytes32(0), deadline,
approvals);
-        _safeTransfer(token, t, bal);
+        if (amount == 0 || amount > _balanceOf(token)) revert ZeroAmount();
+        _approved(this.sweep.selector, token, t, amount, bytes32(0), deadline,
approvals);
+        _safeTransfer(token, t, amount);
```

Status

Resolved

Centralisation

The InstaSwap project values security and utility while limiting what any single privileged party can do alone.

The owner and operator executable functions within the protocol increase security and functionality, and they are constrained on chain. The owner is a 2 of 3 Gnosis Safe, and every loosening change waits 48 hours before it can be executed. During that delay, the owner or a separate guardian Safe, whose signer sits outside the owner Safe, can cancel the change or pause the factory, while the guardian cannot block its own replacement. Every fund moving call and every escrow creation requires 3 of 5 approver signatures, above an immutable floor of 3 that no owner action can lower. According to the InstaSwap team, the five approver signers are operated by three organisations, and none of them reaches the threshold on its own. The treasury is immutable, and the customer refund becomes permissionless after the 24 hour refund timeout.

The remaining trust lies in the custody of the Safe signers and of the approver keys.



Conclusion

After Hashlock's analysis, the InstaSwap project seems to have a sound and well-tested code base, and all of our findings have now been resolved. Overall, most of the code is correctly ordered and follows industry best practices. The code is well commented on as well. Within the scope described in this report and to the best of our ability, Hashlock is not able to identify any further vulnerabilities.

Our Methodology

Hashlock strives to maintain a transparent working process and to make our audits a collaborative effort. The objective of our security audits is to improve the quality of systems and upcoming projects we review and to aim for sufficient remediation to help protect users and project leaders. Below is the methodology we use in our security audit process.

Manual Code Review:

In manually analysing all of the code, we seek to find any potential issues with code logic, error handling, protocol and header parsing, cryptographic errors, and random number generators. We also watch for areas where more defensive programming could reduce the risk of future mistakes and speed up future audits. Although our primary focus is on the in-scope code, we examine dependency code and behaviour when it is relevant to a particular line of investigation.

Vulnerability Analysis:

Our methodologies include manual code analysis, user interface interaction, and white box penetration testing. We consider the project's website, specifications, and whitepaper (if available) to attain a high-level understanding of what functionality the smart contract under review contains. We then communicate with the developers and founders to gain insight into their vision for the project. We install and deploy the relevant software, exploring the user interactions and roles. While we do this, we brainstorm threat models and attack surfaces. We read design documentation, review other audit results, search for similar projects, examine source code dependencies, skim open issue tickets, and generally investigate details other than the implementation.

Documenting Results:

We undergo a robust, transparent process for analysing potential security vulnerabilities and seeing them through to successful remediation. When a potential issue is discovered, we immediately create an issue entry for it in this document, even though we have not yet verified the feasibility and impact of the issue. This process is vast because we document our suspicions early even if they are later shown to not represent exploitable vulnerabilities. We generally follow a process of first documenting the suspicion with unresolved questions, and then confirming the issue through code analysis, live experimentation, or automated tests. Code analysis is the most tentative, and we strive to provide test code, log captures, or screenshots demonstrating our confirmation. After this, we analyse the feasibility of an attack in a live system.

Suggested Solutions:

We search for immediate mitigations that live deployments can take and finally, we suggest the requirements for remediation engineering for future releases. The mitigation and remediation recommendations should be scrutinised by the developers and deployment engineers, and successful mitigation and remediation is an ongoing collaborative process after we deliver our report, and before the contract details are made public.

Disclaimers

Hashlock's Disclaimer

Hashlock's team has analysed these smart contracts in accordance with the best industry practices at the date of this report, in relation to: cybersecurity vulnerabilities and issues in the smart contract source code, the details of which are disclosed in this report, (Source Code); the Source Code compilation, deployment, and functionality (performing the intended functions).

Due to the fact that the total number of test cases is unlimited, the audit makes no statements or warranties on the security of the code. It also cannot be considered as a sufficient assessment regarding the utility and safety of the code, bug-free status, or any other statements of the contract. While we have done our best in conducting the analysis and producing this report, it is important to note that you should not rely on this report only. We also suggest conducting a bug bounty program to confirm the high level of security of this smart contract.

Hashlock is not responsible for the safety of any funds and is not in any way liable for the security of the project.

Technical Disclaimer

Smart contracts are deployed and executed on a blockchain platform. The platform, its programming language, and other software related to the smart contract can have their own vulnerabilities that can lead to attacks. Thus, the audit can't guarantee the explicit security of the audited smart contracts.

About Hashlock

Hashlock is an Australian-based company aiming to help facilitate the successful widespread adoption of distributed ledger technology. Our key services all have a focus on security, as well as projects that focus on streamlined adoption in the business sector.

Hashlock is excited to continue to grow its partnerships with developers and other web3-oriented companies to collaborate on secure innovation, helping businesses and decentralised entities alike.

Website: hashlock.com.au
Contact: info@hashlock.com.au

