PUBLIC

Code Assessment

of the Superswap Router Smart Contracts

June 30, 2025

Produced for

by

Contents

1 Executive Summary

Dear Velodrome team,

Thank you for trusting us to help Velodrome with this security audit. Our executive summary provides an overview of subjects covered in our audit of the latest reviewed contracts of Superswap Router according to Scope to support you in forming an opinion on their security risks.

Velodrome implements an update to the Uniswap Universal Router contract with changes to support velodrome V2, Concentrate Liquidity Pools (CL), bridging tokens and executing arbitrary cross chain actions.

Our audit focused on critical subjects such as allowances management, integration with xVELO and xERC20 bridges, and interchain account integration for cross-chain actions. Allowance management security was found to be high, as previous concerns regarding arbitrary approvals have been addressed.

In addition, we reviewed general subjects including the correctness of Velodrome pools integration and the general functional correctness of the Router. The security of these general areas was also evaluated to be high, as previous issues with amount calculation for Uniswap V2 have been resolved.

The two notes, Router Allowance Trust Risk and Interchain Account Trust Risk, highlight significant differences in the trust model of the Superswap Router compared to the Uniswap Router. These differences should be carefully considered by users of the router, and when updating the router.

In summary, we find that the codebase provides a high level of security.

It is important to note that security audits are time-boxed and cannot uncover all vulnerabilities. They complement but don't replace other vital measures to secure a project.

The following sections will give an overview of the system, our methodology, the issues uncovered, and how they have been addressed. We are happy to receive questions and feedback to improve our service.

Sincerely yours,

ChainSecurity

1.1 Overview of the Findings

Below we provide a brief numerical overview of the findings and how they have been addressed.

Critical-Severity Findings	0
High-Severity Findings	0
Medium-Severity Findings	2
• Code Corrected	2
Low-Severity Findings	0

2 Assessment Overview

In this section, we briefly describe the overall structure and scope of the engagement, including the code commit which is referenced throughout this report.

2.1 Scope

The assessment was performed on the source code files inside the Superswap Router repository based on the documentation files.

The following directories were included in the scope of the assessment:

- `contracts/base/`
- `contracts/interfaces/`
- `contracts/libraries/`
- `contracts/modules/`
- `contracts/types/`
- `contracts/UniversalRouter.sol`

This audit was a diff audit of the codebase with base commit `8bd498a3fc9f8bc8577e626c024c4fcf0691f885`. This means that in the scope of this assessment, the system at this base commit is trusted to be functional and secure, and only the changes made after this commit are considered for the audit.

The table below indicates the code versions relevant to this report and when they were received.

V	Date	Commit Hash	Note
1	20 April 2025	df14477d2e24dd26e413cc6a52a4f5702c74fd3a	Initial Version
2	20 May 2025	ae1cf680bde32137a1f66e579c7c4dce378e9619	Fixes
3	10 June 2025	96d7469daf0228b895969ef9144b3da63604ae48	Version 3

For the solidity smart contracts, the compiler version `0.8.29` was chosen.

2.1.1 Excluded from scope

Anything that is not explicitly mentioned in the scope section above is excluded from the scope of this audit. This includes, but is not limited to:

- `lib/`
- `contracts/test/`
- `contracts/deploy/`
- `node_modules/`

The system at the base commit is trusted and considered out of scope for this audit.

2.2 System Overview

This system overview describes the initially received version (Version 1) of the contracts as defined in the Assessment Overview.

Furthermore, in the findings section, we have added a version icon to each of the findings to increase the readability of the report.

Velodrome adapts the Uniswap Universal Router contract with changes to support velodrome V2, Concentrate Liquidity Pools (CL), bridging tokens and executing arbitrary cross chain actions. In the following sections, we will provide an overview of the system architecture, followed by a description of the new and updated commands.

2.2.1 Overview of the architecture

The Universal Router is a contract that allows users to execute a sequence of commands in a single transaction. Commands include swapping in Velodrome or Uniswap pools, transferring tokens to and from the caller, bridging tokens, and executing cross-chain actions. The contract is designed to be modular, allowing for easy addition of new commands in the future.

The only entry point is the `execute` function. This function is responsible for executing the sequence of commands passed to it.

```
execute(bytes calldata commands, bytes[] calldata inputs, uint256 deadline)
execute(bytes calldata commands, bytes[] calldata inputs)
```

If a deadline is provided, the transaction will revert if it is not executed before the deadline.

The contract work as a Virtual Machine (VM) that executes the provided sequence of commands, with for each command, specific input parameters.

2.2.1.1 Command Structure

A command is encoded in one byte.

```
0 | 1 2 | 3 4 5 6 7
-------------------
f | r   | command
```

Where:

- `f` is a bit flag that signals whether the command should be allowed to revert without the whole transaction failing.
- `r` are two unused bytes, which are reserved for future use.
- `command` is a 5 bits identifier that represents the command itself according to the following table:

```
0x00: V3_SWAP_EXACT_IN
0x01: V3_SWAP_EXACT_OUT
0x02: PERMIT2_TRANSFER_FROM
0x03: PERMIT2_PERMIT_BATCH
0x04: SWEEP
0x05: TRANSFER
0x06: PAY_PORTION
0x07: TRANSFER_FROM
0x08: V2_SWAP_EXACT_IN
0x09: V2_SWAP_EXACT_OUT
0x0a: PERMIT2_PERMIT
```

```
0x0b: WRAP_ETH
0x0c: UNWRAP_WETH
0x0d: PERMIT2_TRANSFER_FROM_BATCH
0x0e: BALANCE_CHECK_ERC20
0x10: V4_SWAP
0x11: V4_INITIALIZE_POOL
0x12: BRIDGE_TOKEN
0x13: EXECUTE_CROSS_CHAIN
0x21: EXECUTE_SUB_PLAN
```

Any command that is not listed in the table is considered invalid and will cause the transaction to revert.

2.2.1.2 Command Inputs

When calling the `execute` function, the caller must provide a sequence of inputs that correspond to the commands being executed. Each element of the bytes array is an ABI-encoded set of parameters for the command. The order of the inputs must match the order of the commands in the bytes array.

2.2.1.3 Message Sender

As the router is designed to allow reentrancy into itself, the contract cannot depend on `msg.sender` to identify the original caller of a command chain. Instead, the system maintains a reference to the initial caller with the `Lock` contract. Throughout this report, we refer to this address as the `sender`. When referring specifically to the actual `msg.sender` of a call, we will use the terms `caller` or `message sender` for clarity.

2.2.2 Transfer from

The `TRANSFER_FROM` (`0x07`) command enables the router to transfer tokens from the sender to a specified recipient. It accepts the following parameters:

```
address token;     // Address of the token to transfer
address recipient; // Address of the recipient
                   //    - 0x01 for the sender
                   //    - 0x02 for the router itself
uint256 value;     // Amount of tokens to transfer
                   //    - 1<<255 for the sender's entire balance
```

The command attempts to call `token.transferFrom()` to move tokens from the sender to the recipient. This requires the sender to have previously granted the router sufficient allowance. If one of the following conditions occur, the router falls back to using a Permit2 `transferFrom` call:

1. The call to `transferFrom` fails (e.g., insufficient allowance provided by the sender).
2. The call succeeds but either:
    - The returned data decodes to `false`, or
    - The returned data cannot be ABI-decoded to a boolean.

In such cases, if the sender has approved the router via Permit2 and have enough funds, the router will utilize the Permit2 allowance to complete the transfer. Otherwise, the transaction will revert.

This command is a more versatile version of `PERMIT2_TRANSFER_FROM` (`0x02`), as it supports both normal `ERC20.transferFrom` on top of Permit2 and the special case where `value == 1<<255`, allowing the transfer of the sender's entire balance.

2.2.3 VeloV2 pools support

The universal router previously supported Uniswap V2 pools through the `V2_SWAP_EXACT_IN` (`0x08`) and `V2_SWAP_EXACT_OUT` (`0x09`) commands. The updated version of the system now includes support for Velodrome V2 pools.

Both commands now include an additional boolean argument, `isUni`, which determines whether the swap path should use Uniswap or Velodrome V2 pools.

Velodrome V2 pools can be categorized as either `stable` or `volatile`, making their paths more complex compared to Uniswap paths. Depending on the value of `isUni`, the logic has been adapted to decode routes accordingly:

- For Uniswap:

```
token0 || token1 || token2 || ... || tokenN
```

- For Velodrome:

```
token0 || is_stable || token1 || is_stable || token2 || ... || tokenN
```

Additional logic adjustments include:

- **Pool Address Computation**: Depending on the value of `isUni`, the computation uses either the Uniswap factory address and Uniswap pool's initcode hash or the Velodrome factory address and Velodrome pool's initcode hash.
- **Fee Handling**: For Uniswap, the fee is fixed at 0.3%. For Velodrome, the fee is dynamically fetched from the factory contract.
- **Output Amount Calculation**: Velodrome stable pools use a different invariant compared to Uniswap pools. The logic has been updated to accommodate this new invariant.

2.2.4 Slipstream pools support

The universal router previously supported Uniswap V3 pools through the `V3_SWAP_EXACT_IN` (`0x00`) and `V3_SWAP_EXACT_OUT` (`0x01`) commands. The updated version of the system introduces support for Velodrome Slipstream pools, which have an interface similar to Uniswap V3 pools.

Both commands now include an additional boolean argument, `isUni`, which specifies whether the swap path should be done through Uniswap V3 pools or Velodrome Slipstream pools.

The main change in the swap logic is the computation of the pool's address for a given token pair. Depending on the value of `isUni`, the computation uses either the Uniswap V3 factory address and Uniswap pool's initcode hash or the Velodrome Slipstream factory address and Velodrome pool's initcode hash. For Uniswap, the computation relies on the `fee` parameter, while for Velodrome, it uses the `tickSpacing` parameter.

The `data` passed to `pool.swap()` is also modified to include the `isUni` flag, which help to ensure that the callback's caller is the correct pool.

2.2.5 Bridging tokens

The `BRIDGE_TOKEN` (`0x12`) command allows the router to bridge tokens to another chain. It accepts the following parameters:

```
uint8 bridgeType;  // Type of bridge to use
address recipient; // Address of the recipient on the target chain
                   //   - 0x01 for the sender
address token;     // Address of the token to bridge
```

```
address bridge;    // Address of the bridge contract
uint256 amount;    // Amount of tokens to bridge
uint256 msgFee;    // Fee to be paid to the bridge contract
uint32 domain;     // Domain ID of the target chain
bool payerIsUser;  // Indicates if the payer is the sender or the router
```

At the time of writing, the supported bridges are as follows:

```
0x01 - Hyperlane xERC20 Bridge
0x02 - XVELO Bridge
```

For both bridges, if `payerIsUser` is true, the router first transfers the tokens to itself. It then approves the bridge contract to spend the tokens. The logic for each bridge is detailed below.

2.2.5.1 Hyperlane xERC20 Bridge

In the Hyperlane xERC20 bridge case, the bridge contract is a HypXERC20 contract. This contract handles the bridging of a specific xERC20 token. If necessary, the user should wrap the token beforehand.

Interaction in the Universal Router:

The router interacts with the bridge by calling `bridge.transferRemote()` with the following parameters:

- The destination domain (a Hyperlane domain ID)
- The recipient's address
- The amount to be bridged
- `msgFee` as `msg.value`
- The sender as the refund address
- The default hook provided by the bridge
- The default destination gas limit provided by the bridge

Sending the message:

Uppon call, the Hyperlane's `HypXERC20` bridge burns the xERC20 tokens from the `UniversalRouter`, and then formats a message for the Hyperlane Mailbox contract. The message includes:

- The recipient address (To which the remote bridge will mint the corresponding amount of xERC20 tokens)
- The token amount being bridged
- Metadata (empty bytes in this case)

If a router for the destination domain was registered by the bridge owner, the message is dispatched through using the `dispatch` function of the Mailbox contract, with `msgFee` as `msg.value`. If no router is registered, the transaction reverts. The destination router should be the corresponding `HypXERC20` instance on the destination chain. The Mailbox contract formats the message containing:

- The destination domain
- The recipient address (the address of the remote `HypXERC20` contract)
- The message body

The Mailbox hook is called to quote the fee required for dispatching the message. The smaller of the quoted fee and the actual `msg.value` sent by the HypXERC20 contract is used. If the value is insufficient, the hook should revert. The `postDispatch` function is called for both the required hook and

the hook from the HypXERC20 contract. Any remaining `msg.value` not sent to the first hook is forwarded to the second.

Relaying the message to the destination:

The relayer receives the dispatched message and calls the `process` function on the destination domain. This function extracts the recipient address, which is the `HypXERC20` contract in this domain. A two-step process is performed, first verifying the message, and then letting the `HypXERC20` contract handle it.

To verify the message, the Mailbox query the `HypXERC20` for its `interchainSecurityModule`. If none is defined, the default Interchain Security Module is used. The Mailbox then calls `verify()` on the selected Interchain Security Module. The verification should ensure the message is genuine.

In case the verification succeeded, the message is marked as delivered, and the `HypXERC20.handle()` is called. Finally, the HypXERC20 contract mints the specified amount of tokens to the recipient.

2.2.5.2 XVELO Bridge

The XVELO Bridge consists of a `RootTokenBridge` contract on Optimism and `LeafTokenBridge` contracts on other target chains. The `RootTokenBridge` contract accepts ERC20 tokens, wrapping them into xERC20 tokens for bridging. On leaf chains, `LeafTokenBridge` contracts expects tokens to already be xERC20 tokens.

Interaction in the Universal Router:

Tokens are bridged by calling `bridge.sendToken()` with the following parameters:

- `msgFee` as `msg.value`
- The recipient's address
- The amount to be bridged
- The destination domain
- The sender as the refund address

Sending the message:

If the source chain is Optimism, the `RootTokenBridge` contract encodes a message containing:

- The recipient address (for the bridged token)
- The token amount

The `RootTokenBridge` contract transfers tokens from the `UniversalRouter` to itself, deposits them in the xERC20 Lockbox to mint xERC20 tokens. These tokens are immediately burned, and the message is dispatched to through the `dispatch` function of the Mailbox contract. The following parameters are passed to the `Mailbox`:

- `fee` as `msg.value` forwarded from the `UniversalRouter`.
- The destination domain
- The recipient address (`LeafTokenBridge` address, same as `RootTokenBridge`)
- The message body
- The metadata
- The `RootTokenBridge` hook

If the `msg.sender` (in this case the `UniversalRouter`) is in the sponsoring whitelist, the bridge covers the Mailbox fee.

The `dispatch` function in the Mailbox contract works similarly to the Hyperlane case.

Relaying the message to the destination:

On the destination chain, a relayer calls the `process` function on the Mailbox in the destination domain. After verification with the `ISM`, The `handle` function of the `LeafTokenBridge` ensure the caller is the `RootTokenBridge` and mints the bridged amount of xERC20 tokens to the recipient.

Bridging from a Leaf chain to the Root chain:

The bridging process from a `LeafTokenBridge` to a `RootTokenBridge` is similar, with the following differences:

- transaction sponsorship is not possible
- and only xERC20 tokens can be bridged. When the token reaches the `RootTokenBridge`, it is converted to its ERC20 counterpart by withdrawing from the Lockbox and sent to the recipient.

2.2.6 Execute cross chain actions

The `EXECUTE_CROSS_CHAIN` (`0x13`) command enables the router to perform cross-chain actions using Hyperlane's Interchain Account Router (ICA) functionality. It accepts the following parameters:

```
uint32 domain;        // Hyperlane domain ID of the target chain
address icaRouter;    // Address of the local ICA router
address remoteRouter; // Address of the remote ICA router
bytes32 ism;          // Address of the Interchain Security Module (ISM) to use
bytes32 commitment;   // Commitment of the action to be executed
uint256 msgFee;       // Fee to be paid to the bridge
address hook;         // Address of the hook contract to be called by the mailbox
bytes hookMetadata;   // Metadata to be passed to the hook contract
```

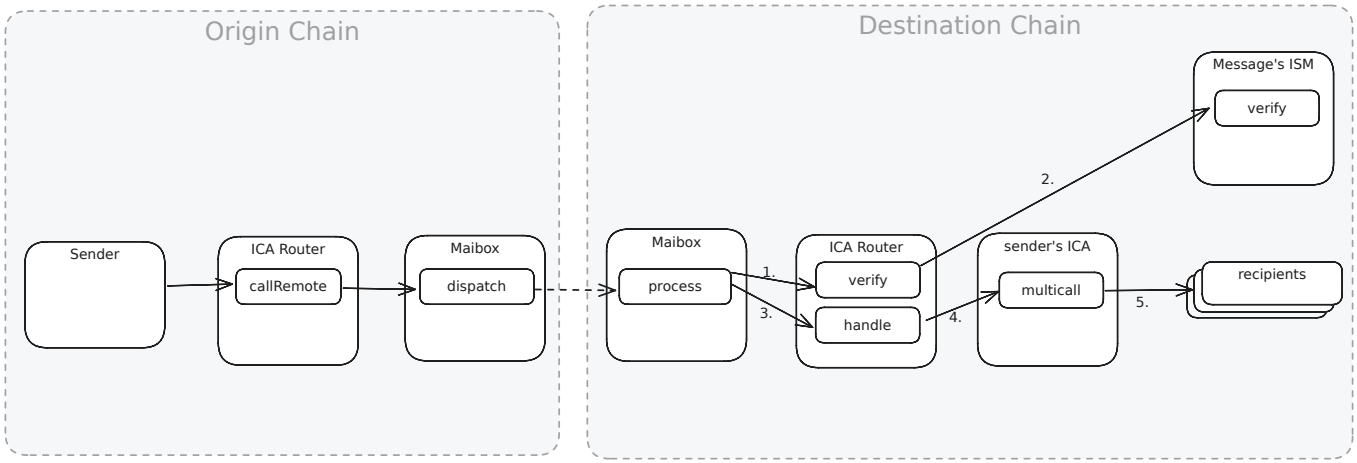
The universal router has no restrictions on these parameters and simply forwards the call to the `icaRouter`, using the sender as the `salt`.

2.2.6.1 Interchain Account

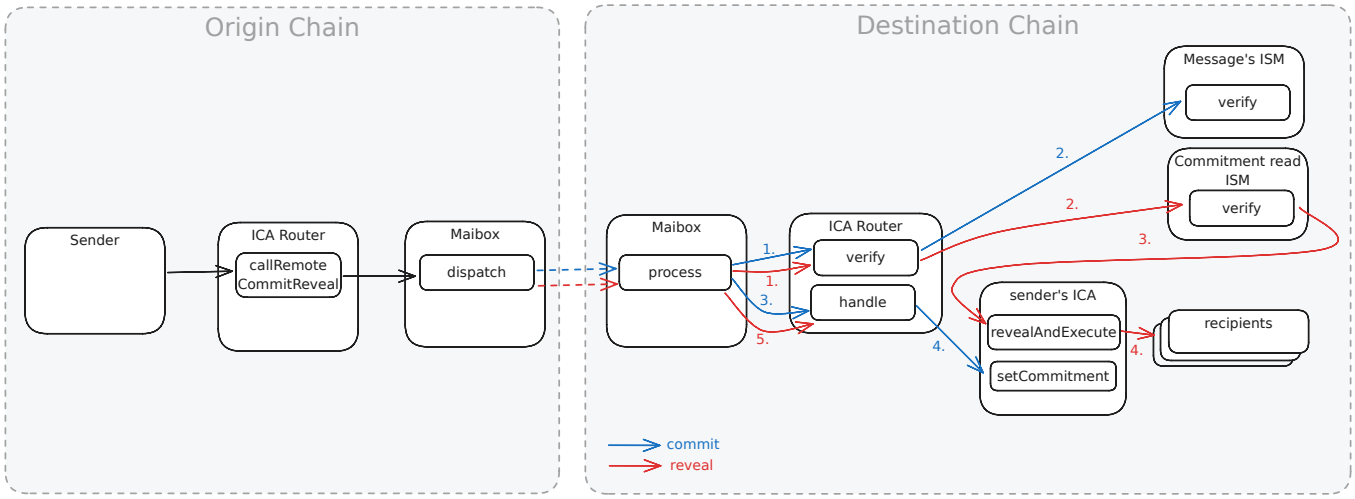
Interchain Accounts are built on top of the Hyperlane protocol and enable cross-chain execution of arbitrary calls on the destination chain.

The Interchain Account Router (ICA Router) is the main contract that manages Interchain Accounts. It is deployed on multiple chains, with each instance communicating with others through the Hyperlane protocol.

Cross chain "direct" execution



Cross chain delayed execution with commitment



Users interact with the ICA Router in two main ways:

Sending a Cross-Chain Action

From the source chain, users can call `callRemote`, `callRemoteWithOverrides`, or `callRemoteCommitReveal` to send a cross-chain action. The following parameters must be provided:

- The `domain` of the destination chain.
- The call(s) to perform on the destination chain (or a commitment to be revealed later).
- The remote `icaRouter` address (optional; if none is provided, a default one for the destination domain is used).
- A hook to be called by the mailbox on the source chain (optional; a default one can also be used but is not mandatory).
- A salt to generate the Interchain Account address (optional; a default one will be used).
- Metadata to be passed to the hook contract on the source chain (optional).

Two types of messages can be sent:

- **Calls**: Sent using `callRemote` or `callRemoteWithOverrides`. The message contains the call data to be executed on the destination chain. The ICA Router encodes the message of type `CALLS` and sends it to `Mailbox.dispatch()`.
- **Commitments**: Sent using `callRemoteCommitReveal`. Two messages are sent: one for the commitment and one for the reveal.

- The first message, of type `COMMITMENT`, contains the commitment (salted hash of the calldata) to be revealed later.
- The second message, of type `REVEAL`, does not contain the calldata but instead includes the address of a CCIP Read ISM contract to be called on the destination chain, along with the commitment. (If none is provided, the default one is used on the destination chain.)

Relaying a Cross-Chain Action

Once the action is sent, it must be relayed to the destination chain. As with any Hyperlane message, the action must be relayed through `Mailbox.process()`, which, after verification by the ICA Router's ISM, calls `icaRouter.handle()`.

The `handle` function first retrieves the Interchain Account from the message. The address of this account is generated using `create2` with the following salt:

```
keccak256(
    abi.encodePacked(_origin, _owner, _router, _ism, _userSalt)
);
```

Where:

- `_origin` is the domain of the source chain.
- `_owner` is the address of the sender.
- `_router` is the address of the ICA Router on the source chain.
- `_ism` is the address of the ISM passed in the message.
- `_userSalt` is the (optional) salt passed in the message by the sender.

This ensures that the Interchain Account is unique to any combination of the above parameters. If the account does not exist, it will be created at that point. An Interchain Account is a simple ownable multicall contract, owned by the ICA Router.

There are three types of messages that can be received by the router:

- **CALLS**: Sent using `callRemote` or `callRemoteWithOverrides`. The message contains the call data to be executed on the destination chain. The ICA Router calls `interchainAccount.execute()` with the calls to be executed.
- **COMMITMENT**: Sent using `callRemoteCommitReveal`. The message contains only a commitment to the calls to be executed, which must be revealed on the destination chain. In this case, the computed Interchain Account stores the commitment for later use during the reveal process.
- **REVEAL**: When revealing a message, the pre-image of the commitment (call data and salt) is passed to `Mailbox.process()` as metadata.

    1. The mailbox forwards the metadata to the `icaRouter`'s ISM through `verify()`. In that case, the ISM is the `icaRouter` itself.
    2. The `icaRouter` forwards the metadata to the CCIP Read ISM, which verifies the pre-image and calls `ica.revealAndExecute()` with it.
    3. `icaRouter.handle()` in that case does nothing.

2.2.6.2 Application of the ICA to the Superswap Router

The Superswap Router leverages the ICA Router to execute cross-chain actions, using the commitment-reveal mechanism to mitigate frontrunning.

When the `EXECUTE_CROSS_CHAIN` command is called, the Superswap Router interacts with the `icaRouter`. The `userSalt` provided is the address of the Superswap Router's sender, ensuring that the Interchain Account is uniquely tied to both the Superswap Router and the sender.

Using the commitment-reveal mechanism, the Superswap Router sends both the `COMMIT` and `REVEAL` messages to the destination chain via the mailbox. Meanwhile, the sender is required to supply the pre-image of the commitment to an off-chain Velodrome Gateway.

Messages received by the mailbox on the source chain are processed by the off-chain Hyperlane relayer, which performs the following actions:

- **Commit Message**: The relayer calls `Mailbox.process()` on the destination chain to verify the message and store the commitment.
- **Reveal Message**:

    1. The relayer calls `CCIPReadISM.getOffchainVerifyInfo()` to retrieve the off-chain location of the commitment's pre-image.
    2. It fetches the pre-image from the Velodrome Gateway.
    3. Finally, it calls `Mailbox.process()` on the destination chain, passing the pre-image as metadata.

Once the reveal process is complete, the command is executed on the destination chain.

2.2.7 Changes in Version 2

In Version 2 of the system, no significant changes were made, instead fixes for the findings of this report were applied.

2.2.8 Changes in Version 3

In Version 3 of the system, the `BRIDGE_TOKEN` command was updated to use the special value `CONTRACT_BALANCE` (`1<<255`) to bridge from the payer, a value equal to the balance of the token in the router.

2.3 Trust Model

The Universal Router has no permissioned roles and is not upgradeable. It is the responsibility of the caller to ensure that the command sequence to be executed is valid and does not lead to a loss of funds. Any funds left in the router at the end of a transaction can be withdrawn by anyone; users should hence be careful not to leave any funds in the router.

Any external dependencies (Hyperlane) are expected to be secure and are trusted, the Hyperlane features used in the router are not yet developed, the Hyperlane dependency is expected to be bumped to a newer version in the future and the interface of Hyperlane should be reviewed accordingly then.

Bridging Tokens:

- Bridges are out of scope of the audit; they are expected to be secured and work as intended. If a bridge is compromised, users' funds can be stolen.
- Tokens are expected to be bridged only to EVM-compatible chains using the `BRIDGE_TOKEN` command.
- For `XVELO`, the root bridge is deployed on Optimism, and the leaf bridges are deployed on other EVM-compatible chains.
- The `gasFee` is paid by the sender, and the transaction will revert if the fee is not sufficient.
- The bridging command is expected to interact only with ERC-compliant tokens that are not malicious and present no specific risks or unusual behaviors (e.g., rebasing, transferring a different

amount than requested, fee-on-transfer, double entry points, non-compliant interface, or hooks). Their xERC20 counterpart is expected to follow the same rules.

Execute Cross-Chain:

- Hyperlane's Interchain Account (ICA) system is expected to be secured and work as intended. If the system is compromised, users' funds can be stolen.
- Cross-chain messages are expected to be sent only to EVM-compatible chains using the `EXECUTE_CROSS_CHAIN` command.
- The xVELO bridge allows whitelisted addresses to have their fees covered by the system. However, it should be noted that a whitelisted address will not receive sponsorship when using the xVELO bridge through the Superswap Router if the latter is not whitelisted.
- Any parties involved in relaying cross-chain transactions are trustworthy (e.g., the relayer and the payload gateway storing the pre-image of commitments).
- Although funds may remain in users' ICA, this is not expected behavior.
- Commitments are expected to be salted with fresh and random values, if the salt is predictable, a commitment can be revealed by a third party and the calls can be frontrun.

Allowances and ICA Router Salt:

- Users are expected to approve the router (directly or through Permit2) to spend their tokens. The security of this mechanism relies on the router never spending tokens from an address that is not the sender. If this assumption is violated, the router can be tricked into spending tokens from an address that is not the sender. This is detailed in Router Allowance Trust Risk.
- Similarly, with the ICA Router, the Superswap Router is expected to only provide as salt the address of the message sender. If not, the router can be tricked into allowing access to the Interchain Account of another address. This is detailed in Interchain Account Trust Risk.

3 Limitations and use of report

Security assessments cannot uncover all existing vulnerabilities; even an assessment in which no vulnerabilities are found is not a guarantee of a secure system. However, code assessments enable the discovery of vulnerabilities that were overlooked during development and areas where additional security measures are necessary. In most cases, applications are either fully protected against a certain type of attack, or they are completely unprotected against it. Some of the issues may affect the entire application, while some lack protection only in certain areas. This is why we carry out a source code assessment aimed at determining all locations that need to be fixed. Within the customer-determined time frame, ChainSecurity has performed an assessment in order to discover as many vulnerabilities as possible.

The focus of our assessment was limited to the code parts defined in the engagement letter. We assessed whether the project follows the provided specifications. These assessments are based on the provided threat model and trust assumptions. We draw attention to the fact that due to inherent limitations in any software development process and software product, an inherent risk exists that even major failures or malfunctions can remain undetected. Further uncertainties exist in any software product or application used during the development, which itself cannot be free from any error or failures. These preconditions can have an impact on the system's code and/or functions and/or operation. We did not assess the underlying third-party infrastructure which adds further inherent risks as we rely on the correct execution of the included third-party technology stack itself. Report readers should also take into account that over the life cycle of any software, changes to the product itself or to the environment in which it is operated can have an impact leading to operational behaviors other than those initially determined in the business specification.

4 Terminology

For the purpose of this assessment, we adopt the following terminology. To classify the severity of our findings, we determine the likelihood and impact (according to the CVSS risk rating methodology).

- *Likelihood* represents the likelihood of a finding to be triggered or exploited in practice
- *Impact* specifies the technical and business-related consequences of a finding
- *Severity* is derived based on the likelihood and the impact

We categorize the findings into four distinct categories, depending on their severity. These severities are derived from the likelihood and the impact using the following table, following a standard risk assessment procedure.

Likelihood	Impact		
	High	Medium	Low
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

As seen in the table above, findings that have both a high likelihood and a high impact are classified as critical. Intuitively, such findings are likely to be triggered and cause significant disruption. Overall, the severity correlates with the associated risk. However, every finding's risk should always be closely checked, regardless of severity.

5 Open Findings

In this section, we describe any open findings. Findings that have been resolved have been moved to the Resolved Findings section. The findings are split into these different categories:

- Design: Architectural shortcomings and design inefficiencies
- Correctness: Mismatches between specification and implementation

Below we provide a numerical overview of the identified findings, split up by their severity.

Critical-Severity Findings	0
High-Severity Findings	0
Medium-Severity Findings	0
Low-Severity Findings	0

6 Resolved Findings

Here, we list findings that have been resolved during the course of the engagement. Their categories are explained in the Open Findings section.

Below we provide a numerical overview of the identified findings, split up by their severity.

Critical-Severity Findings	0
High-Severity Findings	0
Medium-Severity Findings	2

- Arbitrary Approval Can Be Obtained From the Router Code Corrected
- Incorrect AmountIn Calculation Causes Swap Failures With Uniswap V2 Code Corrected

Low-Severity Findings	0
Informational Findings	7

- Ambiguous Sender Parameter in BridgeRouter.prepareTokensForBridge Code Corrected
- Ambiguous CONTRACT_BALANCE Constant Code Corrected
- Duplicate Function _f and _k Code Corrected
- Incorrect Path Length Condition Code Corrected
- Incorrect Recipient Emitted in Dispatcher Events Code Corrected
- Outdated Dependencies Code Corrected
- Potential Gas and Code Size Optimization Code Corrected

6.1 Arbitrary Approval Can Be Obtained From the Router

Design Medium Version 1 Code Corrected

CS-VELO-SR-001

In the Superswap router contract, it is possible to obtain an arbitrary approval from the router contract to transfer out any amount of any token. If a malicious actor manages to be called by the router as part of the transaction of a honest actor, they can use this approval to transfer any amount of tokens left in the router by the sender, in the middle of their transaction.

Obtaining an Arbitrary Approval from the Router:

In the `BridgeRouter` contract, before calling the bridge contract, a token approval is given to the bridge contract itself to allow it to pull the tokens from the router.

```
function prepareTokensForBridge(address _token, address _bridge, address _sender, uint256 _amount, address _payer)
    private
{
    if (_payer != address(this)) {
        payOrPermit2Transfer({token: _token, payer: _sender, recipient: address(this), amount: _amount});
    }
    ERC20(_token).safeApprove({to: address(_bridge), amount: _amount});
}
```

Given that neither the token nor the bridge contract is verified, this allows anyone to give themselves an approval to transfer any amount of any token from the router.

For example, one could deploy a contract with the same interface as the xERC20 bridge but with a dummy `transferRemote()` function that does nothing, and call the Superswap router with the bridge command:

- For a token often used in the router, like USDC
- With a very large amount
- `payerIsUser` set to false

This allows the attacker to get an arbitrary approval to transfer any amount of USDC from the router contract to their own contract.

Exploiting the Allowance

In several instances, the router can call arbitrary addresses with unbounded gas, for example with the command `TRANSFER`:

```
function pay(address token, address recipient, uint256 value) internal {
    if (token == Constants.ETH) {
        recipient.safeTransferETH(value);
    } else {
        ...
    }
}
```

This means that if an honest actor uses the `TRANSFER` command or any such command triggering a call to an arbitrary address with the attacker as the `recipient`, the attacker can use the approval they obtained to transfer any amount of tokens that were left in the router to themselves.

Note that this attack vector is new, as before, the router would never give approval to untrusted recipients. Additionally, it was and is not possible to reenter the router itself to transfer out the funds due to the reentrancy lock held by the honest actor.

Code corrected:

Approvals are reset to zero after the external call to the bridge contract to remove dangling approvals.

6.2 Incorrect `AmountIn` Calculation Causes Swap Failures With Uniswap V2

Correctness | Medium | Version 1 | Code Corrected

CS-VELO-SR-002

The `getAmountIn()` function calculates insufficient input amounts for Uniswap V2 swaps, resulting in failed swap operations.

The implementation of `getAmountIn` deviates from the standard Uniswap V2 formula in its calculation methodology, leading to precision loss and insufficient input amounts.

```
function getAmountIn(uint256 fee, uint256 amountOut, uint256 reserveIn, uint256 reserveOut, bool stable)
    internal
    pure
    returns (uint256 amountIn)
```

```
{
    if (reserveIn == 0 || reserveOut == 0) revert InvalidReserves();
    if (!stable) {
        amountIn = (amountOut * reserveIn) / (reserveOut - amountOut);
        amountIn = amountIn * 10_000 / (10_000 - fee) + 1;
    } else {
        revert StableExactOutputUnsupported();
    }
}
```

The function performs the calculation in two separate steps:

1. First calculates the amount without considering the fee
2. Then applies the fee adjustment as a secondary operation

This approach introduces compound rounding errors due to multiple integer division operations, whereas the standard Uniswap V2 formula combines these calculations:

```
function getAmountIn(uint256 amountOut, uint256 reserveIn, uint256 reserveOut)
    internal
    pure
    returns (uint256 amountIn)
{
    if (reserveIn == 0 || reserveOut == 0) revert InvalidReserves();
    uint256 numerator = reserveIn * amountOut * 1000;
    uint256 denominator = (reserveOut - amountOut) * 997;
    amountIn = (numerator / denominator) + 1;
}
```

Example:

For an Uniswap V2 pool with reserves:

- `reserve0 = 9900000000003045425`
- `reserve1 = 9900000000000000000`

And `amountOut = 49499999999999999092`

The incorrect calculation yields `99297893681046180333` which is insufficient for the swap.

Code corrected:

In Version 2, the `getAmountIn` function was updated to use the original Uniswap formula for calculating amounts when interacting with Uniswap pools (`isUni == true`).

6.3 Ambiguous Sender Parameter in `BridgeRouter.prepareTokensForBridge`

Informational | Version 1 | Code Corrected

CS-VELO-SR-003

In the `prepareTokensForBridge` function of the `BridgeRouter` contract, when `_payer` is not the contract, the `payOrPermit2Transfer` call uses the `_sender` parameter as the payer instead of the `_payer` parameter:

```
function prepareTokensForBridge(address _token, address _bridge, address _sender, uint256 _amount, address _payer)
    private
{
    if (_payer != address(this)) {
        payOrPermit2Transfer({token: _token, payer: _sender, recipient: address(this), amount: _amount});
    }
    ERC20(_token).safeApprove({to: address(_bridge), amount: _amount});
}
```

Although in current usage `_payer` always equals `_sender`, using `_sender` here is misleading and prevents removing the now-redundant `_sender` parameter.

Code corrected:

The `_sender` parameter was removed and the `payOrPermit2Transfer` call was updated to use the `_payer` parameter instead.

6.4 Ambiguous CONTRACT_BALANCE Constant

Informational | Version 1 | Code Corrected

CS-VELO-SR-004

In the `dispatch` function of the Dispatcher contract, within the `Commands.TRANSFER_FROM` branch, when `value` equals `ActionConstants.CONTRACT_BALANCE`, the code retrieves the token balance of the payer (`msgSender()`) rather than the contract address, which is correct but misleading given the constant name.

Code corrected:

In Version 2, a new constant named `TOTAL_BALANCE` was added to the Constants contract and is being used in the `TRANSFER_FROM` command handler. This constant is an alias of `ActionConstants.CONTRACT_BALANCE`.

6.5 Duplicate Function _f and _k

Informational | Version 1 | Code Corrected

CS-VELO-SR-005

In the `UniswapV2Library`, the `_k` and `_f` functions are identical.

Code corrected:

The `_k` function was removed, and its usage was replaced with the `_f` function.

6.6 Incorrect Path Length Condition

Informational | Version 1 | Code Corrected

CS-VELO-SR-006

In the `getAmountInMultihop` function of the `V2SwapRouter` contract, multi-hop Velodrome paths (when `isUni` is `false`) are validated using the Uniswap implementation of `v2HasMultipleTokens`, which checks:

```
function v2HasMultipleTokens(bytes calldata path) internal pure returns (bool) {
    return path.length >= Constants.V2_MULTIPLE_TOKENS_MIN_LENGTH; // ADDR_SIZE * 2
}
```

However, Velodrome path segments include an extra boolean (indicating stable pools) between each token pair, so the minimum length should be `ADDR_SIZE * 2 + 1`. Using the Uniswap threshold causes invalid Velodrome multi-hop paths to be accepted.

Code corrected:

In Version 2, the `hasMultipleRoutes()` function was introduced to handle Velodrome paths, the function `getAmountInMultihop()` was refactored accordingly.

6.7 Incorrect Recipient Emitted in Dispatcher Events

Informational | Version 1 | Code Corrected

CS-VELO-SR-007

In the `dispatch` function of the Dispatcher contract, the `UniversalRouterSwap` event emit the `recipient` value. However, this value is taken directly from the decoded input instead of using the value returned by `map(recipient)`. This occurs in the following command branches:

- `Commands.V3_SWAP_EXACT_IN`
- `Commands.V3_SWAP_EXACT_OUT`
- `Commands.V2_SWAP_EXACT_IN`
- `Commands.V2_SWAP_EXACT_OUT`

Code corrected:

The event was updated to use the value returned by `map(recipient)` instead of the raw value from the decoded input.

6.8 Outdated Dependencies

Informational | Version 1 | Code Corrected

CS-VELO-SR-009

The following dependencies are outdated:

- `@hyperlane-xyz/core`: currently using `5.12.0`, latest available is `7.1.4`
- `@openzeppelin/contracts`: currently using `5.0.2`, latest available is `5.3.0`
- `@uniswap/v3-core`: currently using `1.0.0`, latest available is `1.0.1`

Code corrected:

The dependencies were updated to the latest versions.

6.9 Potential Gas and Code Size Optimization

Informational | Version 1 | Code Corrected

CS-VELO-SR-010

Below is a non-exhaustive list of potential gas and code size savings:

- In the `_veloSwap` function of the `V2SwapRouter` contract, the first route segment is decoded twice by calling `routes.veloRouteAt(0).veloDecodePair()` before the loop and `routes.veloRouteAt(0).decodeRoute()` inside the loop when `i == 0`.
- In the `pairAndToken0For` function of the `V2SwapRouter` contract, the call to `sortTokens` is duplicated in each of the two branches, it could be moved outside the if/else statement.

Code corrected:

In Version 2, the `_veloSwap` function reuses the `input`, `output`, and `stable` variables instead of re-declaring them. The `pairAndToken0For` function calls `sortTokens()` once outside the if/else block, avoiding duplication.

7 Informational

We utilize this section to point out informational findings that are less severe than issues. These informational issues allow us to point out more theoretical findings. Their explanation hopefully improves the overall understanding of the project's security. Furthermore, we point out findings which are unrelated to security.

7.1 `TOKEN_BRIDGE` Always Uses the Balance of the Router

Informational **Version 3** **Acknowledged**

CS-VELO-SR-011

In the `Dispatcher` contract, the `BRIDGE_TOKEN` command is used to bridge tokens from the router to a recipient. The command allows for a special value `CONTRACT_BALANCE` (`1<<255`) to be used as the amount, which indicates that the entire balance of the token in the router should be bridged.

```
address sender = msgSender();
address payer = payerIsUser ? sender : address(this);
recipient = recipient == ActionConstants.MSG_SENDER ? sender : recipient;
if (amount == ActionConstants.CONTRACT_BALANCE) amount = ERC20(token).balanceOf(address(this));
```

However, if the `payerIsUser` flag is set to true, the command will use the sender's address as the payer, but still use the router's balance as the amount to bridge. This might lead to unexpected behavior depending on the user's balance and approval for the token.

Acknowledged:

The Velodrome team has acknowledged this finding and answered:

```
The behaviour aligns with the existing behaviour of the UniversalRouter.
```

7.2 Missing Payer Check in `v3SwapExactInput`

Informational **Version 1** **Acknowledged**

CS-VELO-SR-008

In the `v3SwapExactInput` function of the `V3SwapRouter` contract, when `amountIn == ActionConstants.CONTRACT_BALANCE`, the token balance is always read from `address(this)` without confirming that the `payer` is the contract:

```
if (amountIn == ActionConstants.CONTRACT_BALANCE) {
    address tokenIn = path.decodeFirstToken();
    amountIn = ERC20(tokenIn).balanceOf(address(this));
}
```

Acknowledged:

Velodrome acknowledged this informational issue.

8 Notes

We leverage this section to highlight further findings that are not necessarily issues. The mentioned topics serve to clarify or support the report, but do not require an immediate modification inside the project. Instead, they should raise awareness in order to improve the overall understanding.

8.1 Interchain Account Trust Risk

Note Version 1

Users not only grant the Universal Router unlimited token allowances (see Router Allowance Trust Risk), but also implicitly place trust in it to manage their interchain accounts on each destination chain.

The primary safeguard for funds in an Interchain Account is that the salt is hard-coded to the `msgSender()` when invoking the Interchain Account Router:

```
IInterchainAccountRouter(icaRouter).callRemoteWithOverrides{value: msgFee}({
    _destination: domain,
    _router: remoteRouter,
    _ism: ism,
    _callsCommitment: commitment,
    _hookMetadata: hookMetadata,
    _salt: TypeCasts.addressToBytes32(msgSender()),
    _hook: IPostDispatchHook(hook)
});
```

However, similarly to the risks described in Router Allowance Trust Risk, if an attacker could manipulate the salt parameter (e.g., through a hash collision or function name reuse), they might send a transaction to the Interchain Account Router that enables them to take control of the Interchain Account. This would allow the attacker to execute arbitrary transactions on the destination chain, including transferring funds out of the Interchain Account.

For instance, if the ICA Router implements a function named `transferFrom` or a function with a selector colliding with `transferFrom`, an attacker could potentially use the `TRANSFER_FROM` command to invoke `router.transferFrom()`. In such a scenario, it might be possible to provide a custom `salt` not tied to the caller, thereby gaining access to the remote Interchain Accounts of other users.

In the current implementation, no such collision was found, but this risk should be carefully evaluated with every modification made to the router.

8.2 Router Allowance Trust Risk

Note Version 1

Users of the Universal Router must grant it token allowances for each token they trade. In practice, frontends often grant infinite allowances (either directly or via Permit2) and do not automatically revoke them. This effectively entrusts the router with unlimited token allowances.

The router allows calls to `transferFrom` and enables execution of external calls to arbitrary contracts with unchecked calldata. If an attacker manipulates the router into calling a function that consumes the allowance of another user, they could potentially drain his funds.

While the current implementation does not expose any direct way to exploit another user's allowance, two realistic risk scenarios exist. These should be carefully evaluated with every modification to the router:

1. **Unrestricted transferFrom Usage**:

   Any call that consumes ERC-20 allowances granted to the router (e.g., `transferFrom`) must enforce that the `from` argument always equals `msgSender()`. If this check is missing or bypassed, a malicious user could exploit this to drain another user's allowance.

2. **Function Selector Collisions in External Calls**:

   The router is trusted by users not to call ERC-20 functions that consume their allowances unless they are the caller.

   However, the router makes unchecked external calls to arbitrary contracts using fixed function selectors. This could be exploited if a function selector collision occurs between such a call and **any** ERC-20 contract.

   Such collisions could arise from identical function selectors (hash collisions), bridge contracts reusing ERC-20 function names, or exotic ERC-20 implementations with matching functions.

   External calls where both the target contract address and function parameters are user-controlled, such as `sendToken` in `executeXVELOBridge`, are particularly vulnerable to such collisions.

   For example:

```
ITokenBridge(bridge).sendToken{value: msgFee}({
    _recipient: recipient,
    _amount: amount,
    _domain: domain,
    _refundAddress: sender
});
```

   In this scenario, if **any** ERC-20 token contract used with the router implements a `sendToken` function, or if a selector collision occurs, an attacker could pass a token contract as `bridge` and craft calldata to invoke its `sendToken` function using the user's allowance, thereby draining their tokens.

Recommendations:

- In future versions of the router, it should be ensured that any function call consuming allowances validates the `from` address.
- Calls with user-controlled targets should be ensured not to collide with standard ERC-20 methods.
- Users of exotic ERC-20 tokens should be aware of the risks associated with matching functions, as these could lead to allowance drainage.