Zellic

December 8, 2023

Prepared for
Martin Rieke
Chainflip Labs

Prepared by
Syed Faraz Abrar
Filippo Cremonese
Aaron Esau
SeungHyeon Kim
Zellic

Chainflip Backend

Substrate Pallet Security Assessment

Contents

About Zellic

Zellic was founded in 2020 by a team of blockchain specialists with more than a decade of combined industry experience. We are leading experts in smart contracts and Web3 development, cryptography, web security, and reverse engineering. Before Zellic, we founded perfect blue ↗, the top competitive hacking team in the world. Since then, our team has won countless cybersecurity contests and blockchain security events.

Zellic aims to treat clients on a case-by-case basis and to consider their individual, unique concerns and business needs. Our goal is to see the long-term success of our partners rather than simply provide a list of present security issues. Similarly, we strive to adapt to our partners' timelines and to be as available as possible. To keep up with our latest endeavors and research, check out our website zellic.io ↗ or follow @zellic_io ↗ on Twitter. If you are interested in partnering with Zellic, please contact us at hello@zellic.io ↗.

1. Executive Summary

Zellic conducted a security assessment for Chainflip Labs from November 6th to December 8th, 2023. During this engagement, Zellic reviewed Chainflip Backend's code for security vulnerabilities, design issues, and general weaknesses in security posture.

1.1. Goals of the Assessment

In a security assessment, goals are framed in terms of questions that we wish to answer. These questions are agreed upon through close communication between Zellic and the client. In this assessment, we sought to answer the following questions:

- Are liquidity-provider assets safe while stored in the liquidity pool?
- Are liquidity providers guaranteed the ability to withdraw collateral associated with unfilled orders?
- Are there possible DOS vectors that could be used against the pallets?
    - Is there any way to inflate storage usage?
    - Is there any way to perform overly resource-intensive computations?
    - Is it possible to trigger a panic, assertion, or analogous error that would cause a DOS?
- Is the off-chain engine safe from possible DOS attacks that would prevent the state chain from witnessing external events or from broadcasting outgoing transactions?
- Is external events witnessing implemented securely?
    - Is the state chain guaranteed to witness every event exactly once?
    - What happens if the multi-sig voters get rotated while a call is being proposed?
- Are outgoing transactions safe in the event of a malfunctioning or malicious engine instance?
- Are there any integer truncation or precision issues?

1.2. Non-goals and Limitations

We did not assess the following areas that were outside the scope of this engagement:

- Engine's multi-sig
- Engine's P2P
- On-chain contracts
- Other out-of-scope components

Due to the time-boxed nature of security assessments in general, there are limitations in the coverage an assessment can provide.

1.3. Results

During our assessment on the scoped Chainflip Backend pallets, we discovered two findings, both of which were of low impact. We have documented one critical finding 3.3. ↗ that Chainflip Labs independently identified during the assessment period to ensure comprehensive coverage in the report.

Additionally, Zellic recorded its notes and observations from the assessment for Chainflip Labs's benefit in the Discussion section (4. ↗) at the end of the document.

Breakdown of Finding Impacts

Impact Level	Count
Critical	1
High	0
Medium	0
Low	2
Informational	0

2. Introduction

2.1. About Chainflip Backend

Chainflip is a cross-chain asset-exchange protocol. At its core are two major innovations:

1. Fully distributed and permissionless 100-of-150 multi-signature vaults using the FROST signing protocol.
2. A novel and highly capital-efficient just-in-time (JIT) AMM.

This allows transfer and exchange of assets across chains without the need for wrapped tokens or trusted intermediaries.

2.2. Methodology

During a security assessment, Zellic works through standard phases of security auditing, including both automated testing and manual review. These processes can vary significantly per engagement, but the majority of the time is spent on a thorough manual review of the entire scope.

Alongside a variety of tools and analyzers used on an as-needed basis, Zellic focuses primarily on the following classes of security and reliability issues:

Basic coding mistakes. Many critical vulnerabilities in the past have been caused by simple, surface-level mistakes that could have easily been caught ahead of time by code review. Depending on the engagement, we may also employ sophisticated analyzers such as model checkers, theorem provers, fuzzers, and so on as necessary. We also perform a cursory review of the code to familiarize ourselves with the pallets.

Business logic errors. Business logic is the heart of any smart contract application. We examine the specifications and designs for inconsistencies, flaws, and weaknesses that create opportunities for abuse. For example, these include problems like unrealistic tokenomics or dangerous arbitrage opportunities. To the best of our abilities, time permitting, we also review the contract logic to ensure that the code implements the expected functionality as specified in the platform's design documents.

Integration risks. Several well-known exploits have not been the result of any bug within the contract itself; rather, they are an unintended consequence of the contract's interaction with the broader DeFi ecosystem. Time permitting, we review external interactions and summarize the associated risks: for example, flash loan attacks, oracle price manipulation, MEV/sandwich attacks, and so on.

Code maturity. We look for potential improvements in the codebase in general. We look for violations of industry best practices and guidelines and code quality standards. We also provide suggestions for possible optimizations, such as gas optimization, upgradability weaknesses, centralization risks, and so on.

For each finding, Zellic assigns it an impact rating based on its severity and likelihood. There is no hard-and-fast formula for calculating a finding's impact. Instead, we assign it on a case-by-case basis based on our judgment and experience. Both the severity and likelihood of an issue affect its impact. For instance, a highly severe issue's impact may be attenuated by a low likelihood. We assign the following impact ratings (ordered by importance): Critical, High, Medium, Low, and Informational.

Zellic organizes its reports such that the most important findings come first in the document, rather than being strictly ordered on impact alone. Thus, we may sometimes emphasize an "Informational" finding higher than a "Low" finding. The key distinction is that although certain findings may have the same impact rating, their *importance* may differ. This varies based on various soft factors, like our clients' threat models, their business needs, and so on. We aim to provide useful and actionable advice to our partners considering their long-term goals, rather than a simple list of security issues at present.

Finally, Zellic provides a list of miscellaneous observations that do not have security impact or are not directly related to the scoped pallets itself. These observations — found in the Discussion (4. ↗) section of the document — may include suggestions for improving the codebase, or general recommendations, but do not necessarily convey that we suggest a code change.

2.3. Scope

The engagement involved a review of the following targets:

Chainflip Backend Pallets

Repository	https://github.com/chainflip-io/chainflip-backend ↗
Version	chainflip-backend: `77da7c1bed9cffaf2c753c816999c27a8d3207d9`
Programs	• state-chain/pallets/cf-lp/* • state-chain/pallets/cf-pools/* • state-chain/pallets/cf-swapping/* • state-chain/pallets/cf-witnesser/* • engine/generate-genesis-keys/src/main.rs • engine/build.rs • engine/src/state_chain_observer/* • engine/src/settings.rs • engine/src/constants.rs • engine/src/main.rs • engine/src/common.rs • engine/src/retrier.rs • engine/src/witness/* • engine/src/witness.rs • engine/src/lib.rs • engine/src/dot/* • engine/src/eth/* • engine/src/btc/* • engine/src/db/* • engine/src/health.rs • engine/src/stream_utils.rs
Type	Rust
Platforms	Substrate, Software

2.4. Project Overview

Zellic was contracted to perform a security assessment with four consultants for a total of eight person-weeks. The assessment was conducted over the course of four calendar weeks.

Contact Information

The following project manager was associated with the engagement:

Chad McDonald
Engagement Manager
chad@zellic.io ↗

The following consultants were engaged to conduct the assessment:

Syed Faraz Abrar
Engineer
faith@zellic.io ↗

Filippo Cremonese
Engineer
fcremo@zellic.io ↗

Aaron Esau
Engineer
aaron@zellic.io ↗

SeungHyeon Kim
Engineer
seunghyeon@zellic.io ↗

2.5. Project Timeline

The key dates of the engagement are detailed below.

November 6, 2023	Kick-off call
November 6, 2023	Start of primary review period
December 8, 2023	End of primary review period

3. Detailed Findings

3.1. Witnesser rotation may queue call multiple times

Target	cf-witnesser		
Category	Coding Mistakes	**Severity**	High
Likelihood	Low	**Impact**	Low

Description

The witnesser pallet is responsible of keeping track of how many witnessers have observed an event as well as dispatching a call to act on the event once enough witnessers have witnessed it. This functionality is implemented by the `witness_at_epoch` function.

The pallet can be paused in case of emergencies via the SafeMode pallet. If the pallet is paused, `witness_at_epoch` does not dispatch the calls immediately. Instead, the calls are scheduled for execution by appending them to a queue (`WitnessedCallsScheduledForDispatch`). The calls will be dispatched by `on_idle` once the safe mode is disabled.

Normally, the same call cannot be executed twice; the pallet keeps track of which calls have executed in each previous valid epoch, and `witness_at_epoch` does not dispatch the call if it has already executed. However, `on_idle` does not perform the same check, so in principle the same call could be dispatched multiple times if it was added to the `WitnessedCallsScheduledForDispatch` queue more than once.

Scenario 1

In a scenario where safe mode is enabled and a witnesser rotation is imminent, the same event could be observed by two different majority sets of validators in two different epochs and therefore be pushed to the queue twice.

Scenario 2

When a call is scheduled, it executes any time there is sufficient weight at the end of the block (`on_idle`).

There is a `force_witness` function that allows the root origin to bypass replay and multi-sig checks and execute a call immediately. This action uses the `dispatch_call` function, which adds the call hash to the `CallHashExecuted` pallet storage; but if a call is already scheduled, it is no longer subject to replay protections.

Impact

In an unlikely edge case scenario, the same call could be scheduled and eventually dispatched twice, duplicating the effects of an event.

Additionally, if a call were scheduled and the `force_witness` function forced the call to be made, it would be replayed. Though the `force_witness` function intends to allow calls to be replayed, the scheduled calls are not intended to be replayed. So, we believe this behavior is unintended.

Recommendations

Ensure the event was not already executed in `on_idle` before dispatching it, similarly to how `witness_at_epoch` performs the check.

Remediation

This issue has been acknowledged by Chainflip Labs, and a fix was implemented in commit 20641113 ↗.

The patch ensures that the `on_idle` function also checks that the call has not been dispatched yet by checking `CallHashExecuted`.

3.2. Seized funds are locked in the protocol

Target	cf-swapping		
Category	Protocol Risks	**Severity**	Medium
Likelihood	Low	**Impact**	Low

Description

Funds can be seized in a few cases.

Swaps

The input asset is seized if its amount is less than the `MinimumSwapAmount`:

```
if amount < MinimumSwapAmount::<T>::get(from) {
    // If the swap amount is less than the minimum required,
    // confiscate the fund and emit an event
    CollectedRejectedFunds::<T>::mutate(from, |fund| {
        *fund = fund.saturating_add(amount)
    });
```

Cross-chain messaging

There are multiple edge-cases where funds can be seized when using cross-chain messaging (CCM).

First, if the CCM call-destination asset's chain is not Ethereum:

```
if ForeignChain::Ethereum != destination_asset.into() {
    return Err(CcmFailReason::UnsupportedForTargetChain)
```

Funds can also be seized if a CCM's deposit amount is not large enough to cover the requested gas budget:

```
} else if deposit_amount < gas_budget {
    return Err(CcmFailReason::InsufficientDepositAmount)
```

Finally, when swapping an asset using a CCM, funds can be seized if the input asset amount is less than the `MinimumSwapAmount`:

```
} else if source_asset != destination_asset &&
    !principal_swap_amount.is_zero() &&
    principal_swap_amount < MinimumSwapAmount::<T>::get(source_asset)
{
    // [...]
    return Err(CcmFailReason::PrincipalSwapAmountTooLow)
```

Funds are seized by accounting them in the pallet storage in `CollectedRejectedFunds`. There is no mechanism to refund users of their seized funds, however the funds may be used to cover fees in some cases. We outline this behavior below.

Impact

Regardless of whether or not this is an issue, we feel the need to document the behavior; however, we do consider this an issue for the following reasons.

How seized funds are used

Seized funds — regardless of how or why they are seized — are left wherever they are transferred to.

In the case of a transfer of Ether to the vault contract, for example, the funds can be used to pay for gas fees.

However, in the case of a transfer of a non-native coin/token (e.g., an ERC-20 such as USDC), the seized amount would be permanently locked in the contract and would not be usable for gas fees or any other purpose.

Situations in which users may lose funds unintentionally

Though intentionally sending funds that do not meet the conditions to swap or CCM — hence causing the funds to be confiscated — would be akin to intentionally sending funds to the `0x0` address (i.e. a footgun), we believe users may also unintentionally lose funds in a few cases including (but not limited to) the following:

- First, a user or application wanting to directly interact with the protocol (i.e., bypassing the dApp and its front-end restrictions) may not be aware of the requirement for a CCM call's destination to be Ethereum. A potentially large amount of assets may be seized in this case.
- Also, users who make deposits cannot atomically commit to swapping and execute their swap. That is, it is possible that — between the time they publish a signed transaction in an Ethereum mempool and the time the swap is recorded on chain (`schedule_swap_from_channel`) and validated to be above the minimum swap

amount (among other checks) — the minimum swap amount configuration could be changed by the governance. It would not be the user's fault that their funds are confiscated.

- It is not unheard of for governance to pass a proposal that unintentionally harms a protocol such as configuring a `MinimumSwapAmount` that is off by a couple orders of magnitude and does not get noticed. Funds may be seized in this case, too.

Finding impact considering severity and likelihood

When evaluating a finding's impact, we first answer two questions:

1. **In the event that the issue happens, regardless of why or how unlikely it is for the issue to happen, how bad would it be in the worst-case scenario?** The possibility for large amount of funds to be seized exists in the case that the CCM destination chain is incorrect. However, it only impacts one user's funds. So, we rate this as **Medium severity**.
2. **How likely is this issue to be exploited or triggered in practice?** Realistically, users would not intentionally trigger this issue as they are deincentivized to do so, of course; however, it is possible for users to unintentionally encounter this issue as described above (poor timing with governance configuration changes, when building on the protocol or bypassing the dApp, etc.). Because the possibility exists for this issue to be unintentionally triggered — though it may be unlikely — we consider this to be **Low likelihood**, our lowest likelihood rating.

Combining these two criteria, we calculate the **Low impact**.

Recommendations

Provide a mechanism to refund (or retrieve) seized funds. Note that, whatever the mechanism is, it is critical that the user whose funds are seized pays the gas fees for retrieval to prevent denial-of-service attacks; if the seized funds are enough to be worth retrieving, the user will pay gas fees for them (or, otherwise, the user will not receive their funds).

We understand that such a feature is not trivial to safely implement. We respect Chainflip Labs's position, described below, considering the Low impact rating of this finding.

Remediation

Chainflip Labs's position on this finding is that the likelihood is low enough to not consider this an issue. Seized native coin (e.g., Ether) can be used for gas fees in the vault contract.

3.3. Broker fees are not taken from swap amount

Target	cf-swapping		
Category	Coding Mistakes	**Severity**	Critical
Likelihood	High	**Impact**	Critical

Note: Chainflip Labs independently discovered and promptly addressed this issue just a few days into the engagement. We commend their proactive approach in identifying and remedying this critical issue. Because the vulnerability exists in the audit version, we document it in this finding for completeness.

Description

The broker commission fee is calculated and added to the accrued fees, but it is not subtracted from the swap amount:

```
fn schedule_swap_from_channel(
    deposit_address: ForeignChainAddress,
    deposit_block_height: u64,
    from: Asset,
    to: Asset,
    amount: AssetAmount,
    destination_address: ForeignChainAddress,
    broker_id: Self::AccountId,
    broker_commission_bps: BasisPoints,
    channel_id: ChannelId,
) {
    let fee = Permill::from_parts(broker_commission_bps as u32
    * BASIS_POINTS_PER_MILLION) *
        amount;

    EarnedBrokerFees::<T>::mutate(&broker_id, from, |earned_fees| {
        earned_fees.saturating_accrue(fee)
    });

    let encoded_destination_address =

    T::AddressConverter::to_encoded_address(destination_address.clone());
    let swap_origin = SwapOrigin::DepositChannel {
```

```
            deposit_address:
        T::AddressConverter::to_encoded_address(deposit_address),
            channel_id,
            deposit_block_height,
        };

        if let Some(swap_id) = Self::schedule_swap_with_check(
            from,
            to,
            amount,
            destination_address.clone(),
            &swap_origin,
        ) {
```

Impact

An attacker could abuse this behavior to completely drain the protocol. Because any user can register as a broker and choose arbitrary fees (up to 100%), any user can be an attacker.

If an attacker chooses 100% fees, then initiates a swap using a deposit channel with X amount of funds, they can receive $2X$ (i.e., double the output) from the following:

- X from the regular egress of a swap (e.g., X USDC for X USDC)
- X from the 100% broker commission rate (once the fees are accounted for the broker but not removed from the swap amount)

By repeatedly performing no-op swaps, an attacker could have enough fees accounted to them to withdraw all of the liquidity from the protocol.

Recommendations

Subtract the `fee` from `amount`.

Remediation

As noted, Chainflip Labs independently identified this issue during the assessment period and remediated it in commit 8aa32486 ↗.

4. Discussion

The purpose of this section is to document miscellaneous observations that we made during the assessment. These discussion notes are not necessarily security related and do not convey that we are suggesting a code change.

4.1. Lack of broker fee limits

Note that in the audit version, there are no constraints on the broker fee limit other than that it is <= 100% (enforced by `Permill`):

```
let fee = Permill::from_parts(broker_commission_bps as u32
    * BASIS_POINTS_PER_MILLION) *
    amount;
```

We do not consider this to be a security issue because users choose their brokers.

4.2. ERC-20 egress transfer reversions result in stuck funds

Note that if an egress ERC-20 transfer reverts for any reason (the receiver is blacklisted, the receiver contract's fallback reverts, etc.), the funds will be permanently locked in the Chainflip protocol.

4.3. Type conversion could lead to fund loss

The Chainflip team is planning to support ERC-20 tokens with a `totalSupply` that falls within the `uint128` range. The current code is mathematically safe, and we are noting this potential issue for the future.

If Chainflip Labs wishes to support ERC-20 tokens that have a `totalSupply` exceeding `uint128`, it is necessary to remove the `value.try_into()` and use the `value` directly (i.e., without conversion). At the moment, the type conversion logic would cause DOS because it would raise an error.

Type conversion to `uint128` (see `engine/src/witness/erc20_deposits.rs#L121`) can lead to fund loss; it depends on the kind of tokens and how the token is implemented. Assume that one of the transfer functions supports `uint256` and the amount of transferred tokens is more than the max of `uint128`. The highest 128 bits will be removed, and the chain will treat it with the wrong amount.

An example where this behavior manifested as an issue can be found on our website ↗.

5. Threat Model

This provides a full threat model description for various functions. A threat model documents a given function's externally controllable inputs and how an attacker could leverage each input to cause harm.

5.1. Pallet: cf-lp

The cf-lp pallet is responsible for handling registration for the liquidity-provider role as well as deposits and withdrawals.

Function: `request_liquidity_deposit_address`

The `request_liquidity_deposit_address` function can be used to request a channel (i.e., an address) where the user can deposit assets to have them credited to their account.

The function ensures that safe mode is not enacted, that the user is a liquidity provider, and that it has a registered liquidity refund address.

It then invokes the deposit handler, implemented by the cf-ingress-egress pallet, to get the details of the deposit channel. Finally, an event containing the details of the deposit channel is emitted.

Function: `withdraw_asset`

The `withdraw_asset` function can be used to withdraw from the available balances of the caller.

The function ensures that safe mode is not enabled and that the user is a liquidity provider. It also validates the destination address where the funds should be withdrawn, ensuring the address can be decoded and that it corresponds to a chain that matches the asset being withdrawn.

If all the above checks pass, the account is debited and a corresponding egress is scheduled by invoking the egress handler, implemented by the cf-ingress-egress pallet.

Finally, an event containing the details of the scheduled egress is deposited.

Function: `register_lp_account`

The `register_lp_account` function can be used to register an account as a liquidity provider. The function is unpermissioned; therefore, any account can become a liquidity provider.

The function simply checks that the caller has signed the extrinsic and assigns the liquidity-provider role to the corresponding account ID.

Function: `register_liquidity_refund_address`

The `register_liquidity_refund_address` function can be used to register a liquidity refund address.

The function checks that the caller is a registered liquidity provider and that the provided refund address can be validly decoded.

It then inserts the address in the `LiquidityRefundAddress` double map, and it deposits an event that contains the details of the operation.

Note that refund addresses are associated to a tuple of account ID and chain ID; therefore, an account can have multiple refunds addresses associated with it (up to one per supported chain).

5.2. Pallet: cf-pools

The cf-pools pallet is responsible for managing liquidity pools (allowing governance to create, enable/disable, and set fees) as well as managing range and limit orders. While the actual logic that computes swap amounts and updates pool state is in another crate implementing the AMM, this pallet does contain some glue code that calls the JIT AMM to accomplish a single leg of a swap (and also deposits an event that signals a swap has taken place).

Additionally, this pallet is also responsible for periodically buying and burning FLIP tokens using USDC fees collected from the swaps, applying deflationary pressure that increases the value of FLIP.

Function: `update_buy_interval`

The `update_buy_interval` function can be used by governance to set the time interval that determines how often `on_initialize` swaps USDC (collected from fees) for FLIP, applying deflationary pressure.

The function ensures the origin is governance, sanity checks the new buy interval to ensure it is not zero, sets the new buy interval in storage, and emits a corresponding event.

Function: `update_pool_enabled`

The `update_pool_enabled` function can be used by governance to enable or disable a pool.

The function ensures the origin is governance and then sets the pool status as enabled or disabled according to the call arguments. Finally, it emits an event that signals the pool status was updated.

Function: `new_pool`

The `new_pool` function can be used by governance to create a new pool for a pair of assets. After ensuring that the origin of the call is governance and that a pool for the given pair of assets does not already exist, the function creates a new pool using the parameters (fee and initial price) specified by the caller.

Finally, an event signalling the creation of the new pool is deposited.

Function: `update_range_order`

The `update_range_order` function can be used to create or update an existing range order. The function takes the amount that should be added to or subtracted from the range order. It also takes an order ID and a tick range.

If the order ID is associated with an existing order, that order will be updated; in this case, the tick range is optional, and if provided the order will also be moved from the current to the newly specified tick range. If the order ID does not correspond to an existing order, a new order will be created, and therefore the tick range is required.

The function ensures that safe mode is not enabled and that the caller is a registered liquidity provider. If an existing order ID and a new tick range were provided, it moves the existing order to the new tick range.

It then applies the specified update — increasing or decreasing the liquidity associated with the order by the given amount. Note that the increase or decrease can be specified in terms of an amount of assets or in terms of liquidity.

The user balance is updated every time their position is changed, ensuring the balance is always consistent with the position.

Function: `set_range_order`

The `set_range_order` function can be used to create or update an existing range order. It differs from `update_range_order` in that it takes a specific amount of liquidity that the order should be set to, instead of an amount that should be added or subtracted. Similarly to `update_range_order`, it also takes an order ID and a tick range.

If the order ID is associated with an existing order, that order will be updated; in this case, the tick range is optional, and if provided the order will also be moved from the current to the newly specified tick range. If the order ID does not correspond to an existing order, a new order will be created, and therefore the tick range is required.

The function ensures that safe mode is not enabled and that the caller is a registered liquidity provider. If an existing order ID and a new tick range were provided, it moves the existing order to the new tick range.

It then sets the order liquidity to the given amount. As with `update_range_order`, the amount can

be specified in terms of an amount of assets or in terms of liquidity.

The user balance is updated every time their position is changed, ensuring the balance is always consistent with the position.

Function: `update_limit_order`

The `update_limit_order` function can be used to create or update an existing limit order. The function takes the amount that should be added to or subtracted from the limit order. It also receives an order ID and a price level.

If the order ID is associated with an existing order, that order will be updated; in this case, the price level is optional, and if provided the order will also be moved from the current to the newly specified price level. If the order ID does not correspond to an existing order, the price level must be specified, as a new order will be created at the specified level.

The function ensures that safe mode is not enabled and that the caller is a registered liquidity provider. If an existing order ID and a new price level were provided, it moves the existing order to the new price level.

It then updates the order, increasing or decreasing the associated liquidity by the given amount. Note that the increase or decrease can be specified in terms of an amount of assets or in terms of liquidity.

As in the case of the functions handling range orders, the user balance is updated every time their position is changed, ensuring the balance is always consistent with the position.

Function: `set_limit_order`

The `set_limit_order` function can be used to create or update an existing limit order. Similarly to `update_limit_order`, the function takes an order ID and a liquidity size; if the order ID corresponds to an existing order, the existing order is updated. Otherwise, a new order is created. Instead of receiving an increase or decrease, `set_limit_order` takes a specific amount to which the order liquidity should be changed to.

The function ensures that safe mode is not enabled and that the caller is a registered liquidity provider. If an existing order ID and a new tick range were provided, it moves the existing order to the new price level.

It then sets the order liquidity to the given amount. The amount can be specified in terms of an amount of assets or in terms of liquidity.

The user balance is updated every time their position is changed, ensuring the balance is always consistent with the position.

Function: `set_pool_fees`

The `set_pool_fees` function can be used by governance to set the percentage taken as fee.

The function ensures that the origin of the call is governance, validates the provided fee to ensure it is capped to a maximum amount, and updates the pool state to set the new fee. Updating fees requires collecting current fees before updating the fee parameter. Collected fees are credited to the liquidity providers. Finally, the function deposits an event to signal the update to the pool state.

Function: `on_initialize`

The `on_initialize` hook is called on block initialization. The hook periodically swaps USDC from collected fees for FLIP, burning the resulting FLIP. This results in deflationary pressure, increasing the value of FLIP.

Swaps are performed at an interval that can be configured by governance using the `update_buy_interval` function.

5.3. Pallet: cf-swapping

This pallet is responsible for managing swaps and cross-chain messages (CCMs).

Function: `request_swap_deposit_address`

The function allows a broker — a role that anyone can register for — to obtain an address to which deposits trigger an action (a swap or a CCM). The broker may configure the action that the cf-swapping pallet should take upon receiving a deposit to the address.

First, deposits must be enabled in SafeMode. Also, the destination address must be on the same chain as the destination asset and the address must be valid.

If `channel_metadata` is set, the action is a CCM. These may only be sent if Ethereum is the destination chain.

Function: `withdraw`

This function withdraws the broker fees that the broker has earned.

First, the origin must be a registered broker. The destination address to egress the fee assets to must be valid and on the same chain as the fee asset.

The earned fees must also be nonzero.

Function: `schedule_swap_from_contract`

This function simply schedules a swap with user-specified parameters. It is intended to be called by the witnesser.

The destination address must be on the same chain as the destination asset, and it must be a valid address.

If the input asset amount is less than the configured `MinimumSwapAmount` pallet storage value for that asset, the funds are confiscated and locked in the protocol. Otherwise, the swap is scheduled.

Function: `ccm_deposit`

Processes a CCM by calling the internal `on_ccm_deposit` function. This function is intended to be called by the witnesser.

The destination address must be a valid address and be on the same chain as the destination asset.

If the CCM destination is not Ethereum, the principal is less than the gas budget, or the CCM intends to swap but the input asset is less than the minimum amount, it confiscates the assets and locks them in the protocol. This logic is implemented in the `principal_and_gas_amounts` function:

```
let gas_budget = channel_metadata.gas_budget;
let principal_swap_amount = deposit_amount.saturating_sub(gas_budget);

if ForeignChain::Ethereum != destination_asset.into() {
    return Err(CcmFailReason::UnsupportedForTargetChain)
} else if deposit_amount < gas_budget {
    return Err(CcmFailReason::InsufficientDepositAmount)
} else if source_asset != destination_asset &&
    !principal_swap_amount.is_zero() &&
    principal_swap_amount < MinimumSwapAmount::<T>::get(source_asset)
{
    // If the CCM's principal requires a swap and is non-zero,
    // then the principal swap amount must be above minimum swap amount
    required.
    return Err(CcmFailReason::PrincipalSwapAmountTooLow)
}
```

Next, the CCM is determined to be either a swap CCM, which may also carry a message, or a messaging-only CCM (i.e., without a swap). If the source and destination assets are the same or the principal swap amount (after subtracting the gas budget) is zero, the CCM is a messaging-only swap. Otherwise, it is a swap CCM, too.

If the destination asset is not the same as the destination asset's chain's gas asset, the CCM requires an additional swap from the principal asset to the gas asset. This happens independently of whether the CCM intends to carry a swap or be messaging only:

```
let gas_swap_id = if let Some(other_gas_asset) = other_gas_asset {
    let swap_id = Self::schedule_swap_internal(
        source_asset,
        other_gas_asset,
        gas_budget,
        SwapType::CcmGas(ccm_id),
    );
    Self::deposit_event(Event::<T>::SwapScheduled {
        swap_id,
        source_asset,
        deposit_amount: gas_budget,
        destination_asset: other_gas_asset,
        destination_address: encoded_destination_address.clone(),
        origin,
        swap_type: SwapType::CcmGas(ccm_id),
        broker_commission: None,
    });
    Some(swap_id)
} else {
    swap_output.gas = Some(gas_budget);
    None
};
```

If a gas swap is not required and the CCM is messaging only, the CCM is executed immediately. Otherwise, it is scheduled to be executed during block finalization (`on_finalize`):

```
if let Some((principal, gas)) = swap_output.completed_result() {
    Self::schedule_ccm_egress(ccm_id, ccm_swap, (principal, gas));
} else {
    PendingCcms::<T>::insert(ccm_id, ccm_swap);
    CcmOutputs::<T>::insert(ccm_id, swap_output);
}
```

Note that if any swaps raise an error and revert the entire swapping transaction in `on_finalize`, no future CCMs will work because the `on_finalize` function will always revert. We did not identify a way to cause such an error to happen that would permanently cause a denial-of-service issue (lack of liquidity can be quickly fixed by depositing LP).

Function: `register_as_broker`

This function simply registers the caller as a broker, assuming the caller is not already registered as any role, and broker registration is enabled in SafeMode.

Function: `set_minimum_swap_amount`

This function simply sets the `MinimumSwapAmount` pallet storage value. The caller must be governance.

5.4. Pallet: cf-witnesser

This pallet is a 2/3 multi-sig for witnessers to acknowledge that they have observed a call.

Function: `witness_at_epoch`

This is the main entry point for witnessers to acknowledge a call on chain in the multi-sig.

Votes are stored in the `Votes` pallet storage at a specific index for each voter. A voter can only vote once, and once the threshold of voters has been reached, the call is either made using the `dispatch_call` function or the call is scheduled in `WitnessCallsScheduledForDispatch` if the SafeMode pallet specifies not to dispatch the call.

If the `epoch_index` of the witnessed call is less than or equal to the last expired epoch, the witnessed call does not execute. The following code also constrains `epoch_index` to ensure it is less than or equal to the current epoch index:

```
let index = T::EpochInfo::authority_index(epoch_index, &who.into())
    .ok_or(Error::<T>::UnauthorisedWitness)? as usize;
```

Scheduled calls are executed in the `on_idle` if sufficient weight remains at the end of each block and SafeMode permits it.

Function: `force_witness`

Allows any call to be forcefully dispatched by the root, provided that

- The `epoch_index` has not expired.
- The call hash and `epoch_index` have at least one registered vote:

```
ensure!(Votes::<T>::contains_key(epoch_index, call_hash),
    Error::<T>::InvalidEpoch);
```

The function does not check that the hash is not registered in `CallHashExecuted` pallet storage to prevent replays; however, it does insert the hash there to record that the call has been made. So, the function may replay a call, but it prevents other methods of executing a call (directly witnessing and executing, or scheduling — ideally) from executing in the future.

5.5. Component: Engine

The engine component is responsible for observing events from external chains and submitting signed extrinsics to witness them and contribute to the consensus of the state chain. Data relayed from external chains include swap requests as well as information about current gas (or gas-equivalent) cost.

It also observes and acts on events originating from the state chain; these events can be categorized in two broad categories:

- Key management — events that require to handle keygen, signing, or key handover (the latter is only applicable to BTC)
- Transaction broadcast — events that request a transaction to be broadcast to an external chain

Additionally, the engine provides a health-check endpoint and an endpoint that provides various activity metrics.

Finally, the engine is responsible for monitoring the state chain to handle multi-sig ceremonies; it also has a component that handles peer-to-peer communication with other engine instances. These parts of the code were not part of the scope of this audit.

Witnessing and prewitnessing external events

The engine starts separate threads that listen for external events happening on supported chains (BTC, ETH, and Polkadot) and that are relevant to Chainflip operation (i.e., filtering for events that only involve supported assets).

External events are acted on twice; a real-time stream is processed to submit prewitness extrinsics to the state chain. These extrinsics are recorded by the state chain and cause an event to be emitted (which can in turn be observed by other nodes monitoring the state chain). However, prewitnesses do not contribute to forming the consensus of what events occurred on external chains. This is the responsibility of the regular witness extrinsic, which is invoked by the engine when it observes an event from a delayed stream. The delay guarantees the event is final, preventing issues such as block reorganizations, which could for instance enable double spending.

Broadcasting to other chains

The engine starts an event handler with a `match` expression to perform operations for events including (but not limited to)

- Keygen requests
- Signing requests
- Handover requests
- Broadcast requests (assuming the current node is the `nominee`)

If broadcast requests fail, it sends an extrinsic to call `transaction_signing_failure` to indicate that it failed on chain.

Additionally, the engine regularly sends heartbeat extrinsics to the state chain to ensure it knows the node is alive.

Replay protection

We inquiried Chainflip Labs about the possibility of a malicious broadcaster falsely invoking `transaction_signing_failure` after successfully broadcasting a transaction, potentially causing an egress message to take effect multiple times. This specific aspect is handled by code that was not in the scope of this audit.

Chainflip Labs confirmed that egressing transactions are rebroadcast multiple times, and that in some cases the same transaction could also be signed multiple times (this is the case of Ethereum transactions). However, all outgoing transactions have some form of replay protection which adequately protects them from this potential attack.

Specifically, in the case of Ethereum transactions the replay protection is contained in the key nonce (KeyManager.sol ↗ line 287), which is a unique hash. A message with an already-seen key nonce will be rejected. Even if the same transaction data could be signed multiple times, the nonce contained whithin the transaction data never changes, therefore preventing the same egress message from being replayed.

Polkadot uses the account nonce to avoid transaction replay: the chain will reject duplicate nonces.

Bitcoin transactions are natively immune from rebroadcasting issues, since input UTXOs cannot be spent multiple times.

6. Assessment Results

During our assessment on the scoped Chainflip Backend pallets, three findings were identified. Two were of low impact. One critical issue was independently discovered and remediated by Chainflip Labs shortly after the audit began. Chainflip Labs acknowledged all findings and implemented fixes.

6.1. Disclaimer

This assessment does not provide any warranties about finding all possible issues within its scope; in other words, the evaluation results do not guarantee the absence of any subsequent issues. Zellic, of course, also cannot make guarantees about any code added to the project after the version reviewed during our assessment. Furthermore, because a single assessment can never be considered comprehensive, we always recommend multiple independent assessments paired with a bug bounty program.

For each finding, Zellic provides a recommended solution. All code samples in these recommendations are intended to convey how an issue may be resolved (i.e., the idea), but they may not be tested or functional code. These recommendations are not exhaustive, and we encourage our partners to consider them as a starting point for further discussion. We are happy to provide additional guidance and advice as needed.

Finally, the contents of this assessment report are for informational purposes only; do not construe any information in this report as legal, tax, investment, or financial advice. Nothing contained in this report constitutes a solicitation or endorsement of a project by Zellic.