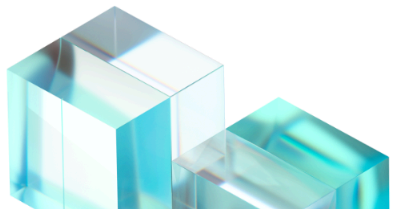
LinumLabs

Frictionless - Audit

Prepared by Linum Labs AG
2025-02-21



Executive Summary

This audit covers the Frictionless order book smart contract system, which supports users buying and selling cryptocurrencies through a traditional order book method.

Protocol Summary

Overview

Frictionless is a platform designed to streamline token swaps through an order book system or "over-the-counter" (OTC) style exchange. Users can create token offers, and contributors can participate by providing an amount in exchange for the offered tokens. The platform leverages Uniswap for price discovery during swaps. It features an innovative fee structure: the more Frictionless tokens a user holds, the lower the fees they pay on their transactions.

Audit Scope

```
../contracts/PrivateController.sol
../contracts/PrivateControllerFactory.sol
../contracts/PrivateSwitch.sol
../contracts/PrivateSwitchFactory.sol
```

Audit Results Summary

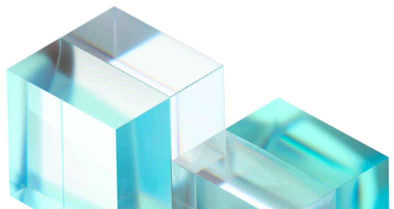
Repository	https://github.com/frictionless-dev/foundry-contracts
Commit	0b682...
Timeline	December 16th - December 29th

Issues Found

Bug Severity	Count
Critical	1
High	1
Medium	4
Low	4
Informational	7
Total Findings	17

Summary of Issues

Description	Severity	Status
Manipulation of offers due to the use of spot prices in the oracle	Critical	Resolved
The last contribution to an offer will always revert	High	Resolved
Restriction on single switch per token	Medium	Resolved
Insufficient validation in *executeDisable()* function	Medium	Resolved
Potential fee manipulation	Medium	Acknowledged
Inconsistent ownership tracking	Medium	Resolved
Use two-step ownership	Low	Resolved



Missing call to *__UUPSUpgradeable_init()* during initialization	Low	Resolved
Redundant owner address validation	Low	Resolved
Compiler version conformation	Low	Resolved
Inconsistent gap sizes	Informational	Acknowledged
Conformance to solidity naming conventions	Informational	Resolved
Unused variable	Informational	Resolved
Unused imports	Informational	Resolved
Missing events	Informational	Resolved
Generic error messages	Informational	Resolved
Improve gas efficiency	Informational	Resolved

Issues

Critical Severity

1. Manipulation of offers due to the use of spot prices in the oracle

Description: The *_tokenToNative()* function in PrivateController.sol relies on real-time Uniswap V2 and V3 pool prices to determine the conversion rate of tokens to native assets. This implementation does not utilize time-weighted average price (TWAP) or other robust oracle mechanisms, making it highly susceptible to price manipulation attacks.

Potential Risk: An attacker can manipulate the pool's spot price by conducting large trades in Uniswap V2 or V3 pools, directly impacting the *_tokenToNative()* calculation output. Such attacks can result in significant losses for the protocol or its users, especially in high-value transactions.

Suggested Mitigation: Replace direct spot price reliance with a TWAP-based or decentralized price oracles like Chainlink. These oracles aggregate price data over time, reducing susceptibility to manipulation.

Frictionless: Fixed in PR

Linum Labs: Verified

High Severity

2. The last contribution to an offer will always revert

Description: In the PrivateSwitch contract, the *_processOffer()* function calculates the *premiumAmount* and adds it to the *amountOut* when verifying the condition:

```Java
amountOut + premiumAmount <= _offer.remainingAmount
```

This approach may lead to an unintended reversion of contributions if the *amountIn* supplied by the *msg.sender* is sufficient to fulfill the entire offer but does not account for the additional premium amount.

Potential Risk: As a result, offers cannot be fulfilled completely, and the function will always revert for edge cases where the *amountOut* exactly matches the *remainingAmount* of the offer.

Suggested Mitigation: Allow for a discounted *amountIn* when a user is the last to contribute. Calculate the needed *amountIn* to reach the required *amountOut (offer.remainingAmount).*

Frictionless: Fixed in PR

Linum Labs: Verified

Medium Severity

3. Restriction on single switch per token

Description: The *createPrivateSwitch()* function in the PrivateController contract currently restricts the creation of multiple switches for the same token. This limitation is enforced by:

```java
require(switches[_tokenAddress] == address(0), "SAE");
```

The above statement will revert if a switch exists for the given *_tokenAddress*.

Potential Risk: This restriction prevents the creation of multiple switches for a token traded on different pools (e.g., TokenA/ETH, TokenA/USDC). Allowing only one switch per token can limit the system's ability to accommodate diverse trading scenarios and potentially hinder its overall functionality. The system is also less adaptable to changes in market conditions or the introduction of new trading pairs involving the same token. Users can still create offers through multi-hop swaps; however, if any configurable variables need to differ between switches, such as premium amounts, it is not possible.

Suggested Mitigation: Change the switches mapping to:

```java
mapping(address => mapping(address => address)) public switches;
```

This mapping will allow the system to store switches based on both the *_tokenAddress* and the *_poolAddress*, enabling the creation of unique switches for each distinct token-pool pair.

Frictionless: Fixed in PR

Linum Labs: Verified

4. Insufficient validation in *executeDisable()* function

Description: The *executeDisable()* function relies on the *disablePendingTimestamp* variable to ensure a time-lock delay is respected before disabling a given switch. However, the function does not validate whether the *proposeDisable()* function is called beforehand to set the *disablePendingTimestamp*. If the *disablePendingTimestamp* is not initialized (i.e., has a default value of 0), the condition checking if the current timestamp is greater than or equal to *disablePendingTimestamp* constantly evaluates as true.

Potential Risk: This allows the *executeDisable()* function to be called directly without following the intended workflow, bypassing the required time lock and enabling immediate switch disabling.

Suggested Mitigation: Ensure that the *executeDisable()* function validates that the *proposeDisable()* function is called for the respective switch before execution. You can achieve this by checking that the *disablePendingTimestamp* is set correctly and is not equal to the default zero value. This validation ensures the time-lock mechanism functions as intended and prevents immediate disabling of a switch.

Frictionless: Fixed in PR

Linum Labs: Verified

5. Potential fee manipulation

Description: The contract calculates user fees by checking the user's frictionless token balance when a contributor calls the relevant function. This design permits contributors to briefly acquire a large frictionless token balance through mechanisms such as flash loans to minimize their fees, after which they dispose of the tokens. This approach undermines any intention to reward long-term token holders since users effectively gain a discount without permanently possessing the tokens.

Potential Risk: Attackers can mitigate their actual fee amount and enable interactions with the contracts for the lowest possible fee.

Suggested Mitigation: A more robust mechanism is advisable so that fee reductions only apply to genuine holders rather than transient beneficiaries. If the existing behavior aligns with project goals, no change is necessary. However, if the aim is to reward actual holders, adjusting the fee logic to prevent instant acquisitions and disposals of tokens is recommended.

Comments: It is, however, noted that on chains such as Ethereum with high gas fees, performing attacks like these would cost significantly more than the savings the attacker would get on the fee. This vulnerability is more prevalent on layer two or roll-ups with much cheaper gas fees.

Frictionless: Frictionless understands the risk of manipulating a user's FRIC token balance to enhance their fee discount and has decided to allow this possibility due to the high throughput it would give their system.

Linum Labs: Acknowledged

6. Inconsistent ownership tracking

Description: The PrivateControllerFactory maintains an *ownerControllers* mapping that tracks which controllers are deployed by a given wallet (owner). The PrivateController contract, which the PrivateControllerFactory deploys, utilizes the OwnableUpgradeable contract, allowing its owner to be changed. However, when the owner of a PrivateController is modified using the OwnableUpgradeable contract's functionality, the *ownerControllers* mapping in the PrivateControllerFactory is not automatically updated, leading to potential inconsistencies in the ownership records.

Potential Risk: Inaccurate Ownership Records: The *ownerControllers* mapping will contain outdated information, leading to incorrect assumptions about which controllers belong to which wallets.

Suggested Mitigation: When the ownership of the PrivateController contract is updated, consider updating the *ownerControllers* mapping in the PrivateControllerFactory contract accordingly. Another solution would be to remove the need for the *ownerControllers* variable in the PrivateControllerFactory and use the subgraph to store the information.

Frictionless: Fixed in PR

Linum Labs: Verified

Low Severity

7. Use two-step ownership

Description: The PrivateController contract inherits from OwnableUpgradeable, granting the contract owner the ability to transfer ownership to any address directly. Additionally, the *setFrictionlessAdmin()* function allows the current *frictionlessAdmin* to transfer their administrative privileges directly to any address.

Potential Risk: A typo in the recipient address or an unexpected event at the recipient address could result in the ownership or *frictionlessAdmin* role being transferred to an unintended or unrecoverable state.

Suggested Mitigation: Consider using a two-step ownership transfer approach instead.

Frictionless: Fixed in PR

Linum Labs: Verified

8. Missing call to *__UUPSUpgradeable_init()* during initialization

Description: The initialize functions in the PrivateControllerFactory and PrivateSwitchFactory contracts do not call the *__UUPSUpgradeable_init()* function from the UUPSUpgradeable contract. While the *__UUPSUpgradeable_init()* function is currently empty and its omission does not introduce a security risk, calling it is a best practice.

Potential Risk: Future updates to the UUPSUpgradeable base contract could introduce logic into its *initializer()* function, and not calling it may result in unexpected issues or inconsistencies. This omission does not adhere to standard initialization procedures for upgradeable contracts.

Suggested Mitigation: Add a call to the *__UUPSUpgradeable_init()* function in the initialize functions of both PrivateControllerFactory and PrivateSwitchFactory. This will ensure that the initialization process adheres to best practices and remains robust against potential future changes to the UUPSUpgradeable contract.

Frictionless: Fixed in PR

Linum Labs: Verified

9. Redundant owner address validation

Description: The *initialize()* function includes a check to ensure the *_owner* parameter is not the zero address. However, this validation is redundant because the *__Ownable_init()* function from OpenZeppelin's OwnableUpgradeable contract already performs the same check. Specifically, the *__Ownable_init()* function internally calls *__Ownable_init_unchained()*, which reverts with OwnableInvalidOwner if the *initialOwner* is the zero address. This unnecessary redundancy increases code complexity without providing additional safety, as the same validation is effectively performed twice.

Potential Risk: This unnecessary redundancy increases code complexity without providing additional safety, as the same validation is effectively performed twice.

Suggested Mitigation: Remove the redundant check from the initialize function.

```Java
require(_owner != address(0), "ISF");
```

Frictionless: Fixed in PR

Linum Labs: Verified

10. Compiler version conformation

Description: The ^ symbol in the Solidity version pragma specifies that the contract can be compiled using any compiler version compatible with the given version, meaning any higher minor or patch version. A version lock allows the developer to define the minimum required compiler version for compiling the contract. This approach ensures compatibility with the compiler features and avoids issues caused by incompatible versions.

Potential Risk: Compiler versions are being released rapidly, with changes that benefit the ecosystem. However, allowing all compiler versions above a specific version risks a new one breaking certain functionality in your contract.

Suggested Mitigation: Locking the Solidity version is recommended to enhance the stability of the codebase. Locking the version helps ensure the contract will not be compiled with an unanticipated compiler version, which could introduce vulnerabilities or bugs. The locked version should be chosen as the lowest possible compiler version that meets the codebase requirements. Doing so allows the project to benefit from a stable, well-tested, long-term support (LTS) environment, reducing potential risks.

Frictionless: Fixed in PR

Linum Labs: Verified

Informational Severity

11. Inconsistent gap sizes

Description: Both the PrivateSwitchFactory and PrivateControllerFactory contracts use a *__gap* array to reserve storage space for future upgrades. However, the total number of storage slots (existing state variables plus the *__gap*) exceeds the typical 50-slot convention recommended by OpenZeppelin. By keeping the total closer to 50, it becomes simpler to track and maintain upgradeable storage layouts.

Potential Risk: Overshooting 50 does not break functionality but can introduce confusion or inconsistency with standard patterns

Suggested Mitigation: Review and adjust each contract's *__gap* array so that the sum of declared storage variables plus the gap slots approximates 50, in line with OpenZeppelin's standard conventions for upgradeable contracts. This helps ensure a uniform and easily maintainable storage layout across upgrades.

Frictionless: This has been acknowledged and will be considered when upgrading. Frictionless will ensure that the gap size is reduced as any more variables are added to the contract.

Linum Labs: Acknowledged

12. Conformance to solidity naming conventions

Description: The Solidity style guide provides best practices for writing clean, consistent, and readable Solidity code. Adhering to these guidelines improves the maintainability and clarity of the code, making it easier for developers to understand and work with. For example, in the PrivateController contract, there are instances where the naming conventions are not adhered to. Specifically, variables such as *FEE_BASE*, *FEE_STEP*, *MAX_STEP*, and *TOKEN_PER_STEP* are written in uppercase, which implies they are constants, but their values can be updated via the *updateFee()* function. This inconsistency may lead to confusion, as the naming suggests immutability when that is not the case.

Potential Risk: This inconsistency may lead to confusion, as the naming suggests immutability when that is not the case.

Suggested Mitigation: The Solidity naming conventions should be followed for better readability and maintainability. For more information, refer to the *Solidity Style Guide*.

Frictionless: Fixed in PR

Linum Labs: Verified

13. Unused variable

Description: The variable address public *owner* is declared and initialized in the *initialize()* function but is not used anywhere in the contract logic. Since it is marked as public, Solidity will automatically generate a getter function for this variable, making it accessible to external users via explorers or direct contract calls. This can lead to confusion, as users might mistakenly assume that the owner variable corresponds to an authoritative role within the contract (e.g., the actual owner or admin of the contract). However, the actual authority is determined by the *controller.owner()* function or other mechanisms, and the owner variable holds no functional purpose.

Potential Risk: Unused variables add unnecessary complexity to the code and may mislead developers or users during contract interactions.

Suggested Mitigation: Remove the *owner* variable if it serves no functional purpose within the contract.

Frictionless: Fixed in PR

Linum Labs: Verified

14. Unused imports

Description: The smart contract contains imports not utilized within the contract's logic.

Potential Risk: Unused imports contribute to contract size, increasing deployment costs, and can make it challenging to understand the dependencies and intended functionality of the contract.

Suggested Mitigation: Remove all unused imports

Frictionless: Fixed in PR

Linum Labs: Verified

15. Missing events

Description: There are external functions, such as *proposeDisable()*, *executeDisable()*, and *cancelDisable(),* in the contract that do not emit any events when they are executed. This is inconsistent with other similar functions in the codebase, such as *addPrivateController()*, *approvePrivateController()*, and *denyPrivateController()*, which emit events to log state changes. Emitting events for external functions is a best practice in Solidity development. Events provide a transparent and immutable record of key state changes, enabling external users, monitoring tools, and off-chain systems to track contract activity.

Suggested Mitigation: Add appropriate events for external functions to ensure transparency and consistency with the rest of the codebase. By adding these events, the contract will maintain consistency with existing functionality and improve traceability.

Frictionless: Fixed in PR

Linum Labs: Verified

16. Generic error messages

Description: Multiple require statements within functions across various contracts utilize the generic error message "ISF" or similar non-descriptive messages.

Potential Risk: The identical error messages hinder debugging and troubleshooting efforts. If a deployment fails, it becomes challenging to quickly determine which dependency contract was not correctly deployed or configured.

Suggested Mitigation: Use more specific error messages to allow users and other developers to understand and debug any problems that may arise efficiently.

Frictionless: Fixed in PR

Linum Labs: Verified

17. Improve gas efficiency

Description: The *cancelOffer()* function in the PrivateSwitch contract cancels an offer by setting the id field of the Offer struct to 0. While this achieves the desired outcome of canceling the offer, it can be more gas-efficient.

Potential Risk: Setting only the id field to 0 does not fully clear the storage slot associated with the offer. This results in unnecessary storage usage and higher gas costs compared to completely removing the offer from storage.

Suggested Mitigation: To optimize the *cancelOffer()* function for gas efficiency, modify it to directly delete the offer from the offers mapping using the delete keyword.

Frictionless: Fixed in PR

Linum Labs: Verified

Disclaimer

This report is based on the materials and documentation provided to Linum Labs Auditing to conduct a security review, as outlined in the Executive Summary and Files in Scope sections. It's important to note that the results presented in this report may not cover all vulnerabilities. Linum Labs Auditing provides this review and report on an as-is, where-is, and as-available basis. By accessing and/or using this report, along with associated services, products, protocols, platforms, content, and materials, you agree to do so at your own risk. Linum Labs Auditing disclaims any liability associated with this report, its content, and any related services and products to the fullest extent permitted by law. This includes implied warranties of merchantability, fitness for a particular purpose, and non-infringement. Linum Labs Auditing does not warrant, endorse, guarantee, or assume responsibility for any third-party products or services advertised or offered through this report, its content, or related services and products. Users should exercise caution and use their best judgment when engaging with third-party providers. It's important to clarify that this report, its content, access, and/or usage thereof, including associated services or materials, should not be considered or relied upon as financial, investment, tax, legal, regulatory, or any other form of advice.