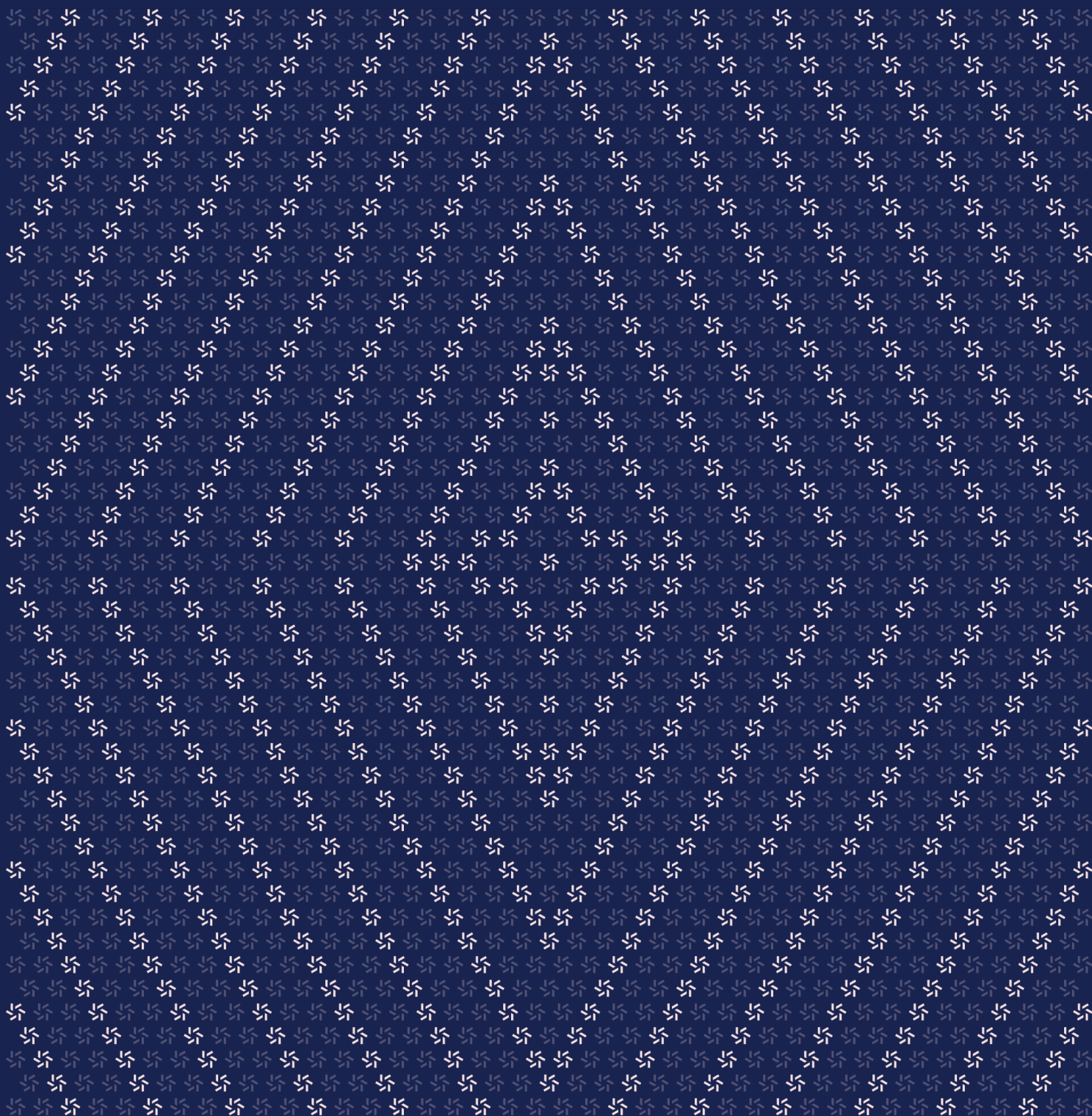


September 18, 2026

Perpshood Smart Contracts

Smart Contract Security Assessment



Contents

About Zellic	4
1. Overview	5
1.1. Executive Summary	5
1.2. Goals of the Assessment	5
1.3. Non-goals and Limitations	5
1.4. Results	5
2. Introduction	7
2.1. About Perpshood Smart Contracts	7
2.2. Methodology	7
2.3. Scope	9
2.4. Project Overview	9
2.5. Project Timeline	10
3. Discussion	11
3.1. The <code>receive</code> function for hypothetical refund should not assume the refund from coordinator	11
4. Threat Model	12
4.1. Module: <code>FlywheelSweepModule.sol</code>	12
4.2. Module: <code>PerpshoodHookrLauncher.sol</code>	14

5.	Assessment Results	22
5.1.	Disclaimer	22

About Zellic

Zellic is a vulnerability research firm with deep expertise in blockchain security. We specialize in EVM, Move (Aptos and Sui), and Solana as well as Cairo, NEAR, and Cosmos. We review L1s and L2s, cross-chain protocols, wallets and applied cryptography, zero-knowledge circuits, web applications, and more.

Prior to Zellic, we founded the [#1 CTF \(competitive hacking\) team](#) [worldwide](#) in 2020, 2021, and 2023. Our engineers bring a rich set of skills and backgrounds, including cryptography, web security, mobile security, low-level exploitation, and finance. Our background in traditional information security and competitive hacking has enabled us to consistently discover hidden vulnerabilities and develop novel security research, earning us the reputation as the go-to security firm for teams whose rate of innovation outpaces the existing security landscape.

For more on Zellic's ongoing security research initiatives, check out our website zellic.io [and](#) follow [@zellic_io](#) [on Twitter](#). If you are interested in partnering with Zellic, contact us at hello@zellic.io [.](#)



1. Overview

1.1. Executive Summary

Zellic conducted a security assessment for Perpshood from September 16th to September 17th, 2026. During this engagement, Zellic reviewed Perpshood Smart Contracts's code for security vulnerabilities, design issues, and general weaknesses in security posture.

1.2. 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:

- Do the contracts in the scope correctly implement the intended behaviors?
 - Could an on-chain attacker interrupt the operation of the contracts in the scope?
-

1.3. Non-goals and Limitations

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

- Front-end components
- Infrastructure relating to the project
- Key custody

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

1.4. Results

During our assessment on the scoped Perpshood Smart Contracts contracts, there were no security vulnerabilities discovered.

Additionally, Zellic recorded its notes and observations from the assessment for the benefit of Perpshood in the Discussion section ([3. 7](#)).

Breakdown of Finding Impacts

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

2. Introduction

2.1. About Perpshood Smart Contracts

Perpshood contributed the following description of Perpshood Smart Contracts:

Perps Hood is a launchpad where every coin is backed by a live leveraged perpetual owned by that coin's own on-chain sub-wallet. Trading fees fund the perp; profits buy back and burn supply; drawdowns only add margin. The position never closes and nothing unwinds at graduation.

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 contracts.

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 contracts itself. These observations — found in the Discussion ([3.7](#)) 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:

Perpshood Smart Contracts Contracts

Type	Solidity
Platform	EVM-compatible
Target	perpshoodfun
Repository	https://github.com/Kolkheti-Labs/perpshoodfun ↗
Version	15bb2851998d43d85983cc31fc8c693620b79182
Programs	FlywheelSweepModule PerpshoodHookrLauncher

2.4. Project Overview

Zellic was contracted to perform a security assessment for a total of two person-days. The assessment was conducted by two consultants over the course of two calendar days.

Contact Information

The following project manager was associated with the engagement:

Pedro Moura
✂ Engagement Manager
pedro@zellic.io ↗

The following consultants were engaged to conduct the assessment:

Jie Ding
✂ Engineer
jie@zellic.io ↗

Jinseo Kim
✂ Engineer
jinseo@zellic.io ↗

2.5. Project Timeline

The key dates of the engagement are detailed below.

September 16, 2026 Start of primary review period

September 17, 2026 End of primary review period

3. 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.

3.1. The `receive` function for hypothetical refund should not assume the refund from coordinator

In `PerpshoodHookrLauncher`, the `receive` function only accepts ETH transfers from the coordinator to handle the refund that might be sent by a future version of the coordinator:

```
/// @dev Only the coordinator may push native here (a hypothetical refund);
    `launch` sweeps it to
/// the user in the same transaction. Anyone else sending ETH is refused.
receive() external payable {
    if (msg.sender != address(coordinator)) revert NotCoordinator();
}
```

Although the relevant contracts do not appear to be upgradeable, this defensive logic is reasonable, as it minimizes possible issues if this contract is deployed alongside a newer coordinator version in the future, or if the coordinator has a bug that incorrectly refunds some amount to the creator.

However, the hypothetical refund might not always come from the coordinator. For example, Hookr's coordinator currently transfers all received ETH to the router, and the router refunds the remaining native tokens to the creator directly (which is not executed during the initial buy).

If it is preferred to have such defensive logic against possible bugs of the coordinator/router/etc. or possible mistakes of future deployers, we recommend loosening the condition where the `receive` function accepts the payment.

In addition, this logic does not guarantee the correctness of this contract after changes on the external contracts; the deployer will have to evaluate such changes and make appropriate changes to this contract to ensure its successful operation.

4. Threat Model

This provides a full threat model description for various functions. As time permitted, we analyzed each function in the contracts and created a written threat model for some critical functions. A threat model documents a given function's externally controllable inputs and how an attacker could leverage each input to cause harm.

Not all functions in the audit scope may have been modeled. The absence of a threat model in this section does not necessarily suggest that a function is safe.

4.1. Module: FlywheelSweepModule.sol

Checked items below indicate manual source review, not necessarily unit-test coverage.

Function: constructor(`_safe`, `_flywheel`)

Sets the immutable Safe and flywheel addresses. The Safe must separately enable the module before it can transfer funds through sweep.

Inputs

- `_safe`
 - **Validation:** Must be nonzero; otherwise, the constructor reverts with `ZeroAddress`.
 - **Impact:** Selects the Safe whose native ETH balance is swept.
- `_flywheel`
 - **Validation:** Must be nonzero; otherwise, the constructor reverts with `ZeroAddress`.
 - **Impact:** Selects the destination of every sweep.

Branches and code coverage

Intended branches

- ☒ Two nonzero addresses are assigned to `safe` and `flywheel`.
- ☒ Both addresses are immutable and cannot be changed after deployment.
- ☒ Deployment does not enable the module on the Safe or transfer funds.

Negative behavior

- ☒ A zero `_safe` reverts before either address is assigned.
- ☒ A nonzero `_safe` and zero `_flywheel` also revert with `ZeroAddress`.

- ☑ A nonzero `_safe` is accepted without checking its code or interface.
- ☑ `_flywheel` may be a contract or equal the Safe or module address; only zero is rejected.

Function call analysis

No function calls are made. The `ISafe` cast does not call or validate `_safe`.

Function: `sweep()`

Allows anyone to forward the Safe's entire native ETH balance to the immutable `flywheel`. The Safe must have enabled the module for a transfer to succeed. The function requests a plain `CALL` with empty `calldata` and returns the amount read before the call.

Inputs

This function has no explicit inputs.

Branches and code coverage

Intended branches

- ☑ A zero Safe balance returns 0 without calling the Safe or emitting an event.
- ☑ A positive balance calls `execTransactionFromModule(flywheel, amount, "", 0)` with the Safe's entire observed balance.
- ☑ A `true` return emits `Swept(amount)` and returns that amount.
- ☑ Every caller uses the same Safe and destination, with empty `calldata` and operation 0. There is no caller-selected amount or retained reserve.

Negative behavior

- ☑ A `false` return reverts with `SweepFailed`, rolling back the call and its subcalls.
- ☑ A Safe revert propagates to the caller. Missing return data or an invalid ABI-encoded `bool` also causes a revert.
- ☑ An EOA configured as the Safe takes the no-op path when empty but cannot complete a positive-balance sweep with the required return data.
- ☑ A recipient rejecting ETH causes a revert if the Safe returns `false` or reverts in response.
- ☑ If the configured recipient or Safe execution path introduces a callback, reentry can trigger another sweep of the then-current Safe balance. The intended `flywheel` EOA does not introduce a recipient callback. Reentry cannot change the immutable Safe or destination, `calldata`, or operation, and there is no mutable accounting to update twice.
- ☑ A `true` return is accepted without checking the Safe's remaining balance or the destination's receipt. This also applies when the destination equals the Safe.

- ☑ Only the Safe's native balance is swept. ETH forcibly sent to the module cannot be withdrawn, and tokens are not swept.

Function call analysis

- `sweep()` -> `safe.execTransactionFromModule(flywheel, amount, "", 0)`
 - **External/internal?:** External call through the ISafe interface.
 - **Argument control?:** The Safe and destination are immutable. The amount is the Safe's balance read at entry; empty calldata and operation 0 are constants. The caller controls timing and can increase the amount by funding the Safe.
 - **Impact:** Requests a native ETH transfer from the Safe to flywheel. The Safe handles module authorization and executes the recipient call. The module sends no ETH with this interface call and uses the returned boolean to decide whether to emit Swept or revert.

4.2. Module: PerpshoodHookrLauncher.sol

Checked items below indicate manual source review, not necessarily unit-test coverage.

Function: constructor(coordinator_)

Sets the immutable coordinator address and initializes the reentrancy guard to 1.

Inputs

- `coordinator_`
 - **Validation:** Must be nonzero. The constructor does not check whether the address contains coordinator code.
 - **Impact:** Selects the destination for market creation and address previews. The address cannot be changed after deployment.

Branches and code coverage

Intended branches

- ☑ A nonzero address is stored as `coordinator`, and `_entered` starts at 1.

Negative behavior

- ☑ A zero address reverts with ZeroAddress.
- ☑ A nonzero address without a compatible coordinator implementation passes construction but can cause subsequent coordinator calls to fail.

Function call analysis

This function makes no internal or external function calls.

Function: intentFor(user, userIntentId)

Returns `keccak256(abi.encode(user, userIntentId))` to namespace coordinator intents by user.

Inputs

- `user`
 - **Validation:** Any address is accepted, including zero. It need not equal the caller.
 - **Impact:** Selects the intent namespace. `launch` supplies `msg.sender`, while `previewToken` supplies its explicit user argument.
- `userIntentId`
 - **Validation:** Any identifier is accepted, including zero and previously used values.
 - **Impact:** Determines the intent within the user's namespace. Repeated-launch rejection is left to the coordinator.

Branches and code coverage

Intended branches

- ☑ The same user and identifier return the same hash.
- ☑ Different users can reuse an identifier without sharing a namespace, subject to hash collision resistance.

Negative behavior

- ☑ Computing an intent does not reserve or consume it. Repeated calls with the same inputs succeed.

Function call analysis

This function makes no internal or external function calls. It uses the `abi.encode` and `keccak256` built-ins.

Function: `previewToken(args, userIntentId, user)`

Returns the coordinator's predicted token address after replacing `expectedCreator` with the launcher and namespacing the intent by `user`.

Inputs

- `args`
 - **Validation:** `expectedCreator` is overwritten with `address(this)`. Metadata, supply, salt, market settings, and initial-buy parameters are otherwise forwarded unchanged. Unlike `launch`, this function does not replace `market.lpFeeRecipient` or reject ERC-20 quotes.
 - **Impact:** Supplies the token and market parameters used by the coordinator's prediction. Agreement with `launch` depends on its address derivation and the arguments ultimately submitted.
- `userIntentId`
 - **Validation:** Any identifier is accepted.
 - **Impact:** Must match the identifier submitted to `launch` for the derived intent to match.
- `user`
 - **Validation:** Any address is accepted; it need not equal the caller.
 - **Impact:** Selects the user namespace for prediction. To match a launch, it must be the account or contract calling `launch`.

Branches and code coverage

Intended branches

- ☒ The function copies the tuple, replaces its creator, derives the intent, and returns the coordinator's response.

Negative behavior

- ☒ A coordinator revert or an undecodable address response reverts the preview.
- ☒ A successful preview does not consume the intent or establish that the market and payment will pass launch validation.

- ☑ Using a different user or changing a prediction-relevant argument can change the expected address.

Function call analysis

- `previewToken -> intentFor(user, userIntentId)`
 - **External/internal?:** Internal pure call.
 - **Argument control?:** Both inputs are caller-supplied.
 - **Impact:** Computes the coordinator-facing intent for the specified user.
- `previewToken -> coordinator.previewNewTokenAddress(a, intentId)`
 - **External/internal?:** External view call to the immutable coordinator.
 - **Argument control?:** The caller controls the tuple except for `expectedCreator`, and supplies the inputs to the derived intent.
 - **Impact:** Computes the prediction under static execution and returns it without further validation.

Function: `launch(args, userIntentId, expectedToken, feeRecipient)`

Allows anyone to open a native-quote token market with the launcher as creator and `feeRecipient` as the founding-position fee recipient. The attached payment goes to the coordinator. The launcher then sends its entire subject-token and remaining native balances to the caller and emits `Launched`.

Inputs

- `args`
 - **Validation:** `expectedCreator` is overwritten with `address(this).name`, `symbol`, `tagline`, `logoURI`, `totalSupply`, and `deploymentSalt` are forwarded unchanged.
 - **Impact:** Controls token metadata, supply, and deployment salt. The coordinator records the launcher as creator, while `Launched.user` records the caller.
- `args.market`
 - **Validation:** quote must be zero, and `lpFeeRecipient` is overwritten with `feeRecipient`. Validation of liquidity amounts, tick spacing, initial price, kernel, module IDs and configurations, and stack limits is left to the coordinator.
 - **Impact:** Defines the proposed market and module stack. The limits cover LP fees, quote and subject takes, module gas, trusted router and quoter, and correction actors, volume, minimum profit, and fee policy. The launcher

does not restrict recipients encoded in module configurations.

- `args.initialBuy`
 - **Validation:** `msg.value` must exactly equal `quoteAmountIn`, including when both are zero. `subjectAmountOutMinimum`, `deadline`, and `moduleData` are forwarded for coordinator and router validation.
 - **Impact:** Controls the initial buy and its execution conditions. The payment check uses the buy amount, not `market.quoteAmount`; the launcher does not independently enforce slippage or expiry.
- `userIntentId`
 - **Validation:** Any identifier is accepted and hashed with `msg.sender`. Replay rejection is delegated to the coordinator.
 - **Impact:** Separates callers' intent namespaces while preserving the same intent for repeated calls by one user.
- `expectedToken`
 - **Validation:** Forwarded unchanged for the coordinator's address check. Zero is documented as skipping that check.
 - **Impact:** Allows the caller to bind the expected token address. The launcher does not compare it with the returned subject.
- `feeRecipient`
 - **Validation:** Must be nonzero; no platform allowlist is enforced.
 - **Impact:** Replaces the nested LP fee recipient and is recorded in `Launched`.

Branches and code coverage

Intended branches

- ☒ The guard changes `_entered` from 1 to 2 before the body and restores it after settlement and event emission.
- ☒ A native-quote launch with a nonzero recipient and exact payment reaches the coordinator with the overwritten creator and recipient, caller-derived intent, and full `msg.value`.
- ☒ A zero initial-buy amount requires zero attached payment and can proceed without a buy.
- ☒ A zero subject balance skips `transfer`; a positive balance is transferred in full to `msg.sender`.
- ☒ A zero native balance skips the caller callback; a positive balance is sent in full with empty `calldata`.
- ☒ `Launched` records the caller, returned subject and pool ID, supplied fee recipient, derived intent, and subject balance measured before transfer.

Negative behavior

- ☑ Reentry into launch from the coordinator, token, or native recipient reverts with Reentrancy. The guard does not block coordinator calls to receive.
- ☑ A zero recipient, ERC-20 quote, or unequal payment reverts before market creation with ZeroAddress, Erc20QuoteUnsupported, or PaymentMismatch, respectively.
- ☑ A coordinator revert or undecodable return tuple aborts the launch. Rejection of duplicate intents, expected-token mismatches, and invalid market configurations is left to the coordinator.
- ☑ A false token-transfer result reverts with SubjectTransferFailed. A reverting or undecodable balanceOf or transfer response also aborts the call; tokens returning no boolean are not supported.
- ☑ A failed native callback reverts with NativeTransferFailed. A launch failure rolls back its market creation, transfers, and guard changes.
- ☑ Settlement includes preexisting balances. Old or forcibly transferred native goes to the next successful caller, and any existing balance of the returned subject is included in subjectOut.
- ☑ Native is measured once. If additional native reaches the launcher during the final callback, it is not included in the measured payment and remains for a later launch.

Function call analysis

- `launch -> intentFor(msg.sender, userIntentId)`
 - **External/internal?:** Internal pure call.
 - **Argument control?:** The caller chooses the identifier; `msg.sender` supplies its namespace.
 - **Impact:** Derives the intent used for coordinator deduplication.
- `launch -> coordinator.openNewTokenMarket{value: msg.value}(a, intentId, expectedToken)`
 - **External/internal?:** External payable call to the immutable coordinator.
 - **Argument control?:** The caller supplies the tuple, expected token, and payment. The launcher replaces the creator and nested recipient and derives the intent from the caller.
 - **Impact:** Creates the token and market and executes the initial buy. The returned subject selects the token used for settlement; the returned pool ID is emitted without further checks.
- `launch -> IERC20Minimal(subject).balanceOf(address(this))`
 - **External/internal?:** External view call.
 - **Argument control?:** The coordinator supplies subject; the account is fixed to the launcher.
 - **Impact:** Reads the whole subject balance used for the transfer and event amount.
- `launch -> IERC20Minimal(subject).transfer(msg.sender, subjectOut)`

- **External/internal?:** External call, skipped for a zero balance.
- **Argument control?:** The coordinator supplies the token, the caller receives it, and `balanceOf` supplies the amount.
- **Impact:** Forwards the reported balance and requires a `true` return. The recipient's balance increase is not measured, so the event amount assumes ordinary ERC-20 transfer behavior.
- `launch -> payable(msg.sender).call{value: native}("")`
 - **External/internal?:** External low-level call with no explicit gas cap, skipped for a zero balance.
 - **Argument control?:** The caller controls the recipient and its code; the amount is the launcher's measured native balance.
 - **Impact:** Sends remaining native and invokes the caller's `receive` or `fallback` function. The guard remains active throughout the callback, and a failed call aborts the launch.

Function: `receive()`

Accepts native transfers from the immutable coordinator. Received funds remain in the launcher until swept by `launch`.

Inputs

This function has no explicit inputs.

Branches and code coverage

Intended branches

- ☒ A coordinator-originated call passes the sender check and accepts any native amount, including zero.
- ☒ Receipts before the launch's native-balance measurement are included in its payment to the caller.

Negative behavior

- ☒ Any other sender reverts with `NotCoordinator`, even when sending zero value.
- ☒ If the coordinator sends native outside a launch or after its native-balance measurement, that balance remains for a later successful launch. Reachability of such payments through the deployed coordinator has not been established.
- ☒ Forced native transfers bypass this function and its sender check.

Function call analysis

This function makes no internal or external function calls.

5. Assessment Results

During our assessment on the scoped Perpshood Smart Contracts contracts, there were no security vulnerabilities discovered.

5.1. 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.

Where Zellic states that a finding has been addressed or fixed in one or more specific commits, this indicates only that those commits introduce changes that, in our assessment, address the finding relative to the codebase version originally reviewed. This does not imply that Zellic reviewed other changes contained in those commits, the overall state of the codebase at those commits, or the interplay between remediation measures for different findings.

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.