Zellic

January 9, 2024

Prepared for
Hasan Atay
Wasabi

Prepared by
Filippo Cremonese
Seunghyeon Kim
Jisub Kim
Zellic

Wasabi Perps

Smart Contract 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 Wasabi from [January 3th to January 8th, 2024]. During this engagement, Zellic reviewed Wasabi Perps'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:

- Is the access control sound and place on the proper functions?
- Is it possible for funds (other than negligible dust) to accrue in the contracts?
- Is it possible for a low privilege or no privilege caller to cause a denial of service by causing the contracts to be in a specific state?
- Is it possible for on-chain attacker to drain arbitrary tokens in the vaults?
- Is it possible to have bad liquidations or debts?
- Does pools and vaults operate as intended?

1.2. Non-goals and Limitations

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

- Owner key management
- Backend and Frontend components

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

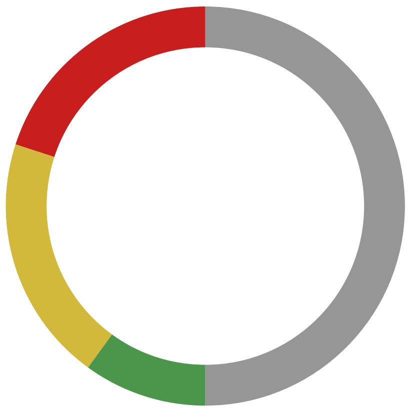
1.3. Results

During our assessment on the scoped Wasabi Perps contracts, we discovered 10 findings. Two critical issues were found. Two were of medium impact, one was of low impact, and the remaining findings were informational in nature.

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

Breakdown of Finding Impacts

Impact Level	Count
🟥 Critical	2
🟧 High	0
🟨 Medium	2
🟩 Low	1
⬜ Informational	5



2. Introduction

2.1. About Wasabi Perps

Wasabi Perps is a asset backed perpetual trading protocol. It uses margin vaults to provide margin loans to users where they can purchase the underlying asset with leverage.

A trader can go long and short with leverage. The leverage is supplied by LPs depositing ETH or ERC20 tokens into vaults. The LPs earn interest in return.

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 (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:

Wasabi Perps Contracts

Repository	https://github.com/DkodaLabs/wasabi_perps ↗
Version	wasabi_perps: `e260b7a1b1898d8f3d8f0fe561927a7f7404571d`
Programs	• AddressProvider • DebtController • WasabiVault • BaseWasabiPool • Hash • PerpUtils • WasabiLongPool • WasabiShortPool
Type	Solidity
Platform	EVM-compatible

2.4. Project Overview

Zellic was contracted to perform a security assessment with three consultants for a total of one person-weeks. The assessment was conducted over the course of one 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:

Filippo Cremonese
Engineer
fcremo@zellic.io ↗

Seunghyeon Kim
Engineer
seunghyeon@zellic.io ↗

Jisub Kim
Engineer
jisub@zellic.io ↗

2.5. Project Timeline

The key dates of the engagement are detailed below.

January 3, 2024	Start of primary review period
January 8, 2024	End of primary review period

3. Detailed Findings

3.1. Centralization risk

Target	WasabiLongPool, WasabiShortPool, WasabiVault, PerpUtils		
Category	Protocol Risks	**Severity**	Critical
Likelihood	N/A	**Impact**	Critical

Description and impact

The design of the contract heavily relies on an off-chain backend component, and it trusts a single centralized owner account. There are multiple implications, which we describe in the following paragraphs.

Lack of privilege separation

Only a single owner authority is defined; this implies that the off-chain backend has access to a private key that can perform routine operations (signing data to authorize a position to be opened or closed), as well as critical operations, such as updating the interest rate or even upgrading the contract. This design increases the impact of a backend compromise to a critical level by default. If the backend private key was compromised, there would be a multitude of ways the contracts could be drained of all their assets, including upgrading the contract, deploying malicious vaults, changing the address of the address provider by invoking `setAddressProvider`.

Basic operation requires centralized signature

Basic operation of the contract, including opening and closing a position, requires parameters signed from the backend. As such, any backend availability issue (for example, due to a software bug, a hardware or connectivity issue, or a denial-of-service attack) would impact the user's ability to use basic functionality, potentially causing a substantial individual and/or collective loss.

Centralized liquidation

Liquidating a position is an action reserved to the owner of the contract. Liquidity providers cannot trigger liquidation of an undercollateralized position and therefore depend on the reliability of the off-chain backend to act in a timely manner.

Ineffective liquidation-threshold checks

The smart contracts attempt to protect their users against a malicious or compromised backend from liquidating a position unfairly. The safety check is implemented by checking that the payout received by the user is less than 5% the amount the user has committed as down payment in case of a short position, or less than 5% of the principal in the case of long positions.

However, since the route used to perform the swap(s) needed to close the position is generated and signed by the backend, a compromised or malicious backend could generate a route that results in a loss for the user, for example by requesting a swap to an entirely different token and/or recipient. Additionally, since the check is based on the spot price, it is likely possible to manipulate the price to liquidate any position without risk, by means of a sandwich attack. Therefore, the on-chain check trying to prevent unfair liquidations is ineffective against a compromised backend.

Antislippage checks

When opening a position, the pools effectively perform an antislippage check after a swap is performed. This check is performed by asserting that the received amount is greater than a minimum amount provided as part of the signed data describing the position being opened. We note that no such check is performed when closing the position. This potentially enables MEV sandwiching attacks when closing or liquidating a position. However, it is assumed that the backend correctly populates the `minAmountOut` field (or alternative equivalent if a market with an API different than Uniswap is used), for all swaps being performed both when opening and when closing a position. At the time of the start of this review, the backend did set antislippage parameters, but there was no option to configure the amount of slippage the user is willing to accept.

Reliance on off-chain–sourced data

The contract performs only limited on-chain enforcement of some security relevant invariants. Some checks are only effective if the signed data describing the position originated by the off-chain backend is correct and was not manipulated by an attacker. For instance, when opening a position, the backend is responsible to provide all the parameters that describe the position being opened, including antislippage parameters, fees, and down payment amount. The parameters are provided as a signed binary string, which is difficult to inspect for correctness.

When opening and closing a position, the data provided by the backend also contains the path to be used to perform the required swaps (collateral for principal or vice versa). However, this functionality is implemented in the most generic way possible, allowing the backend to provide a list of arbitrary addresses that will be called with arbitrary calldata and arbitrary ETH value. While this design choice allows for great flexibility, it also greatly increases the potential attack surface and makes a security analysis much harder.

Recommendations

Implement separate roles for separate operations to limit the impact of a backend compromise. Reduce the dependency on the off-chain backend and add more verification logic on the contracts.

Remediation

Lack of privilege separation

This issue has been acknowledged by Wasabi, and a fix was implemented in commit `73faeca1` ↗.

Wasabi implemented separate roles for the pools, by introducing a new PerpManager contract which tracks the following user roles:

- `LIQUIDATOR_ROLE` is required to liquidate orders
- `ORDER_SIGNER_ROLE` is required to submit new orders
- `ADMIN_ROLE` is required to perform other administrative actions (upgrading contracts, adding vaults)

The team communicated they intend to implement the following key and role management plan.

Only one account will be assigned the admin role; the account will be a Gnosis safe multisig, associated to three or more private keys controlled by members of the development team. The wallet will be configured to require approval from the majority of the controlling private keys to be operated.

The order signer role will be assigned to a separate keypair that will be available to Wasabi backend. No ETH need to be owned by this address. The keypair will be rotated weekly by using the admin keys.

The liquidator role will be assigned to another separate keypair that will be available to a separate internal liquidation server. This server will be separate from the backend and not exposed to the internet. The private key will be rotated bi-monthly.

Basic operation requires centralized signature

Wasabi added a `claimPosition` function to both the long and short pool contracts in commit `183ad5e8` ↗. The function allows to close an existing position without involving the backend. The user is responsible for obtaining the liquidity required to pay back the principal plus interests (and fees, in case of long positions), and for authorizing Wasabi to use `transferFrom` to be able to transfer them from the user balance. The `claimPosition` function pulls the owed amount of principal from the user, and returns the collateral (minus closing fees in the case of short positions) to the user.

Centralized liquidation

Wasabi acknowledged the considerations and risks posed by centralized liquidation. For the time being, liquidation will remain centralized.

Ineffective liquidation threshold checks

Wasabi acknowledged the risks of this design decision; fully mitigating this issue would require a more decentralized design, including use of an independent on-chain price oracle, and limiting the possibility to provide fully custom routes from the backend. Without those changes, any on-chain liquidation threshold check based only on the amount of assets received after the swap route is executed cannot be effective.

Anti-slippage checks

Wasabi added the possibility to specify the acceptable percentage of price slippage to the frontend UI.

Reliance on off-chain sourced data

Wasabi acknowledged the issue, and opted to keep the current centralized design for the time being. They provided the below comment.

> Wasabi's order creation and liquidations are currently controlled by a single address. This allows us to create the most efficient trade orders and provide the best prices for our users while also protecting our LPs. The centralization of the orders help us block malicious users from creating paths that can drain value from Wasabi. Liquidations are currently only done by the team, but will be available to the public in the near future, allowing MEV opportunities while also increasing the overall health of our protocol.

3.2. Incorrect down payment calculation

Target	WasabiShortPool		
Category	Coding Mistakes	**Severity**	Critical
Likelihood	High	**Impact**	Critical

Description

When opening a short position, the short pool contract imposes a limit on the amount of principal that can be borrowed. The limit depends on the amount of collateral that is provided by the user opening the position.

The limit is enforced after the swap of principal for collateral is performed, reverting the transaction (and therefore the swap) if the operation is determined to be overleveraged. The `openPosition` function converts the down payment amount (denominated in terms of the collateral asset) to the corresponding amount of principal that it could buy back, using the same exchange rate at which the borrowed principal was traded for the collateral.

There is a mistake in the formula used to compute the converted down payment amount, in the following line:

```
uint256 swappedDownPaymentAmount = _request.downPayment
    * _request.principal / (collateralReceived - _request.downPayment);
```

The down payment is incorrectly subtracted from `collateralReceived`, which is a variable containing the amount of collateral received from the swap, not including the down payment.

Impact

The incorrect calculation leads to overestimating the down payment, allowing to borrow with more leverage than intended.

Recommendations

Replace the incorrect calculation with the following.

```
uint256 swappedDownPaymentAmount = _request.downPayment
    * _request.principal / collateralReceive);
```

Remediation

This issue has been acknowledged by Wasabi, and a fix was implemented in commit `0a42697f` ↗.

3.3. Unused on-chain interest calculation

Target	WasabiShortPool		
Category	Coding Mistakes	**Severity**	High
Likelihood	High	**Impact**	Medium

Description

When closing a short position, the short pool computes the amount of interest required from the user. However, the result of the calculation is not used.

```
function _closePositionInternal(
    bool _unwrapWETH,
    uint256 _interest,
    Position calldata _position,
    FunctionCallData[] calldata _swapFunctions
) internal returns(uint256 payout, uint256 principalRepaid,
    uint256 interestPaid, uint256 feeAmount) {
    if (positions[_position.id] != _position.hash())
    revert InvalidPosition();
    if (_swapFunctions.length == 0) revert SwapFunctionNeeded();

    _interest = _computeInterest(_position, _interest);
    // [...]
```

The `_interest` variable is not used anywhere in `_closePositionInternal`.

Impact

Ignoring the computed interest removes any on-chain check regarding the amount of interest paid. This means that there is no on-chain enforcement of any upper or lower limit to the interest the user has to pay.

Recommendations

Check that the user has paid the amount of interest computed on chain.

Remediation

This issue has been acknowledged by Wasabi, and a fix was implemented in commit `dacffc05` ↗.

The team implemented checks that ensure the amount of interest repaid is within 3% of the on-chain computed interest. While the 3% margin of error is not necessary, this fix does constrain the amount of interest repaid.

3.4. Zero interest automatically changed to maximum interest

Target	BaseWasabiPool		
Category	Coding Mistakes	**Severity**	Low
Likelihood	Low	**Impact**	Informational

Description

When closing a position, the pools compute the amount of interest owed by the user with the `_computeInterest` function.

```
function _computeInterest(Position calldata _position, uint256 _interest)
    internal view returns (uint256) {
    uint256 maxInterest = addressProvider.getDebtController()
        .computeMaxInterest(_position.currency, _position.principal,
    _position.lastFundingTimestamp);
    if (_interest == 0 || _interest > maxInterest) {
        _interest = maxInterest;
    }
    return _interest;
}
```

We note that due to how the function is programmed, it would be impossible to offer zero interest, as an interest of zero is automatically converted into the maximum possible interest.

Impact

This design renders it impossible to offer zero rates. Additionally, using zero as the special value signalling that maximum interest should be used is arguably more error prone than other alternatives, such as using a separate variable or a value of `uint256.max` (which would automatically be capped to the maximum possible value).

Recommendations

Consider changing the behavior of the function to use `uint256.max` as the special value that signals the maximum possible interest should be applied.

Remediation

This issue has been acknowledged by Wasabi.

3.5. Loss of precision

Target	DebtController		
Category	Coding Mistakes	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

The `DebtController::computeMaxInterest` function computes the maximum interest that a user should be charged given a position size, age, and principal. Currently, the principal address is ignored and only the position size and age are considered.

```
function computeMaxInterest(
    address,
    uint256 _principal,
    uint256 _lastFundingTimestamp
) public view returns(uint256 maxInterestToPay) {
    uint256 secondsSince = block.timestamp - _lastFundingTimestamp;
    maxInterestToPay = _principal * maxApy / APY_DENOMINATOR * secondsSince
    / (365 days);
}
```

The function divides an intermediate result by `APY_DENOMINATOR` before multiplying again by `secondsSince`.

Impact

Dividing before multiplying or adding is generally discouraged as it introduces unneeded roundings due to integer arithmetic. This could result in a slightly lower than intended maximum interest.

Recommendations

Consider changing the computation to divide after all multiplications are done.

```
maxInterestToPay = _principal * maxApy / APY_DENOMINATOR * secondsSince
    / (365 days);
maxInterestToPay = _principal * maxApy * secondsSince / (APY_DENOMINATOR
          * 365 days);
```

Remediation

This issue has been acknowledged by Wasabi, and a fix was implemented in commit `c6d230cd` ↗.

3.6. Missing length check

Target	BaseWasabiPool		
Category	Coding Mistakes	**Severity**	Low
Likelihood	N/A	**Impact**	Informational

Description

The `BaseWasabiPool::liquidatePositions` function is checking that the length of the `_positions` argument equals the length of the `_interests` argument, but it is not checking the length of `_swapFunctions`.

```
function liquidatePositions(
    bool _unwrapWETH,
    uint256[] calldata _interests,
    Position[] calldata _positions,
    FunctionCallData[][] calldata _swapFunctions
) external payable onlyOwner {
    uint256 length = _positions.length;
    if (length != _interests.length) revert InterestAmountNeeded();
    for (uint i = 0; i < length; ++i) {
        liquidatePosition(_unwrapWETH, _interests[i], _positions[i],
    _swapFunctions[i]);
    }
}
```

Impact

There is no security impact, and as such this finding is reported as informational. A mismatching length would just cause a revert. Since this is a function reserved to protocol owners, it has no impact on the user experience. We report this with the purpose of improving the quality and consistency of the codebase.

Recommendations

Consider adding a check on the length of the `_swapFunctions` argument.

Remediation

This issue has been acknowledged by Wasabi, and a fix was implemented in commit `1aee88f1` ↗.

3.7. Initializers not disabled in ownable and upgradable implementation contracts

Target	WasabiVault, BaseWasabiPool		
Category	Coding Mistakes	**Severity**	Low
Likelihood	N/A	**Impact**	Informational

Description

The WasabiVault and BaseWasabiPool contracts are upgradable, inheriting from standard OpenZeppelin UUPSUpgradable and Ownable contracts. Those contracts do not disable initializers in their constructors.

Impact

Due to using a recent version of OpenZeppelin, this issue is not exploitable and there is no security impact, and as such this finding is reported as informational.

In earlier versions of OpenZeppelin, not disabling initializers meant anyone could call the implementation contracts directly and initialize them, usually transferring ownership of the contract. The data associated with the normal operation of the contract is not affected by this, since the storage belongs to the proxy that invokes the implementation contract using `delegatecall`. However, having ownership of the implementation contract allows to invoke and permanently destroy the implementation contract by using `UUPSUpgradable.upgradeToAndCall` to `delegatecall` a contract that performs a `selfdestruct`. This has the effect of permanently breaking the smart contract, since the upgrade functionality is also contained in the now destroyed implementation contract.

Newer versions of OpenZeppelin remediated this issue by implementing an `onlyProxy` modifier used by `upgradeToAndCall`, which reverts if the function is invoked directly on a deployed contract instead of through a proxy.

Recommendations

No action is required.

Remediation

We reported this issue as soon as it was identified due to its potentially critical impact. The Wasabi team promptly developed and submitted a patched version of the contracts, which disabled initializers in the contract constructors and was deployed shortly after.

- January 5th, 18:51 UTC+1 — Issue raised to the development team.
- January 5th, 19:25 UTC+1 — Issue acknowledged.
- January 5th, 19:39 UTC+1 — Patch commit `1ea56e94` ↗ submitted for our review.
- January 5th, 19:41 UTC+1 — Patch commit `1ea56e94` ↗ reviewed.
- January 5th, ~19:45 UTC+1 — Contracts are upgraded.
- January 8th, 11:11 UTC+1 — Wasabi informed that the issue was not exploitable.

3.8. User is able to revert a position being closed

Target	PerpUtils, BaseWasabiPool, WasabiLongPool, WasabiShortPool		
Category	Coding Mistakes	**Severity**	Medium
Likelihood	Medium	**Impact**	Medium

Description

Both long and short pools make use of the common `BaseWasabiPool::_payCloseAmounts` function to transfer the fees and the payout when closing or liquidating a position.

The function takes an `_unwrapWETH` argument that determines whether the transfer should be made as WETH or ETH. If an ETH transfer is requested, the function will use `PerpUtils.payETH` to transfer the required fees and payout amounts to, respectively, the fee-collector address and the user address.

Since reverts are not caught, there is the possibility to prevent a position from being closed or liquidated. If the recipient of the transfer is a smart contract, it is possible for it to revert, either directly or for example by consuming all the available gas.

Impact

This issue allows a malicious user to prevent liquidating a position under the following two conditions:

1. The payout is requested in ETH (`_unwrapETH` is true).
2. The position is undercollateralized but not insolvent (the payout is still greater than zero after fees are deducted).

The latter condition is required because the user address is called only if the value to transfer is greater than zero. Because of this, an attacker cannot prevent liquidation indefinitely, but they can prevent it until their position is insolvent.

Recommendations

The possibility to consume all the gas can be addressed by limiting the amount of gas forwarded by `PerpUtils.payEth`.

However, this does not completely address the issue, as the recipient smart contract could immediately execute a revert. One possibility would be to consider forbidding usage of Wasabi Perps through smart contracts or, somewhat less constraining, forbidding the recipient of the payout to be a smart contract. This is achievable by checking that `extcodesize == 0` for the recipient of the payout. If restricting the recipient of the payout to being an EOA is not acceptable, we recommend supporting only WETH for payouts. Alternatively, it could be possible to ignore reverts originated by the transfer, but this could allow benign smart contract recipients to lose their payout if they reverted for any reason while receiving it.

Remediation

This issue has been acknowledged by Wasabi.

Wasabi informed us that currently liquidations happen without unwrapping WETH. They opted to not make changes to the smart contracts at this time, and proposed a potential off-chain mitigation in case they wanted to liquidate a position while unwrapping WETH: a failed liquidation could be detected and retried without unwrapping WETH.

3.9. Setters lack sanity checks

Target	AddressProvider, DebtController, BaseWasabiPool		
Category	Coding Mistakes	**Severity**	Informational
Likelihood	Low	**Impact**	Informational

Description

The owner-only setters in AddressProvider, DebtController, and BaseWasabiPool lack sanity checks. Some basic checks, such as ensuring an address is not zero or that a numeric parameter is within reasonable bounds, are commonly implemented as a defensive programming practice to increase the resilience of the codebase against unintentional errors.

Impact

This issue does not constitute an exploitable vulnerability, and as such is reported as informational. Implementing basic sanity checks could prevent trivial human errors from setting unintended, clearly incorrect values.

Recommendations

Consider adding basic sanity checks to the setters that lack them.

Remediation

This issue has been acknowledged by Wasabi, and a fix was implemented in commit `5fab86d9` ↗.

3.10. Potential reentrancy issue

Target	WasabiVault		
Category	Coding Mistakes	**Severity**	Informational
Likelihood	Low	**Impact**	Informational

Description

The `WasabiVault::_withdraw` function does not follow the checks-effects-interactions pattern, and contains a potential reentrancy issue.

```
function _withdraw(
    address caller,
    address receiver,
    address owner,
    uint256 assets,
    uint256 shares
) internal virtual override {
    if (caller != owner) {
        _spendAllowance(owner, caller, shares);
    }

    _burn(owner, shares);
    pool.withdraw(asset(), assets, receiver);

    totalAssetValue -= assets;

    emit Withdraw(caller, receiver, owner, assets, shares);
}
```

The `totalAssetValue` variable is updated only after `pool.withdraw(...)` is invoked.

Impact

Assuming only legitimate ERC20 assets are used, and therefore that the attacker cannot gain control during the call to `pool.withdraw(...)`, this issue does not constitute an exploitable vulnerability, and as such is reported as informational.

However, careful vetting of the allowed ERC20 assets is required unless a code change is made to mitigate the issue.

Recommendations

Strictly follow the checks-effects-interactions pattern, updating `totalAssetValue` before invoking `pool.withdraw(...)`.

Consider adding reentrancy protection to the contract, using the `nonReentrant` modifier on all functions that could be exploited if reentered. All public functions should ideally be protected, unless reentrancy is explicitly desired.

Remediation

This issue has been acknowledged by Wasabi, and a fix was implemented in commit `a5b73f2f` ↗.

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. Limited test coverage

The codebase contains a set of testcases partially covering some of the functionality. However, there are noticeable gaps in test coverage that require additional test cases. For example, the short pool is significantly less covered by negative tests with respect to the long pool. Some of the most fundamental functionality, including `openPosition`, appears to be tested only at a superficial level, without stressing possible edge cases.

For a more in-depth understanding of these deficiencies, please refer to (5. ↗), which offers an analysis of some of the missing test cases. Having a robust suite of test cases is crucial to ensure that the code remains resilient and functions correctly, even when subjected to extreme or edge case scenarios. It is important to address these missing tests to maintain high standards of code reliability and integrity.

4.2. The difficulty of analyzing code that relies on offchain components

It is difficult to analyze code that depends on off-chain components providing arbitrary call targets and calldata.

Merely examining the contract code does not give the specific call targets and an array of calldata in use. A comprehensive audit, encompassing both the backend and frontend, is essential to ascertain the details of these call parameters.

5. 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.

5.1. Module: BaseWasabiPool.sol

Function: `withdraw(address _token, uint256 _amount, address _receiver)`

This function can be called by vaults when a user wants to redeem pool tokens in exchange for the corresponding amount of underlying assets.

Inputs

- `_token`
  - **Control**: None (controlled by the vault).
  - **Constraints**: `msg.sender` and `_token` must match according to pool configuration.
  - **Impact**: Asset to be withdrawn.
- `_amount`
  - **Control**: None (controlled by the vault).
  - **Constraints**: None.
  - **Impact**: Amount to be withdrawn.
- `_receiver`
  - **Control**: Arbitrary (user can specify the receiver).
  - **Constraints**: None.
  - **Impact**: Receiver of the withdrawn asset.

Branches and code coverage

Intended branches

- After checking `msg.sender` against `_token`, it transfers the requested amount of assets to the recipient.
  - ☐ Test coverage

Negative behavior

- Reverts if `msg.sender` is not the vault associated with `_token`.
  - ☐ Negative test

Function call analysis

- `SafeERC20.safeTransfer(IERC20(_token), _receiver, _amount)`
    - **What is controllable?** `_receiver`.
    - **If the return value is controllable, how is it used and how can it go wrong?** Not used.
    - **What happens if it reverts, reenters or does other unusual control flow?** Reentrancy is an issue — not exploitable in practice because Wasabi allowlists a set of tokens that do not allow an attacker to reenter on `transfer`.

5.2. Module: DebtController.sol

Function: `computeMaxInterest(address, uint256 _principal, uint256 _lastFundingTimestamp)`

This function computes the maximum interest amount to pay for the loan.

Inputs

- `<unnamed@0>`
    - **Impact**: Unused.
- `_principal`
    - **Control**: Full control.
    - **Constraints**: None.
    - **Impact**: The principal borrowed.
- `_lastFundingTimestamp`
    - **Control**: Full control.
    - **Constraints**: Must be lower than the current `block.timestamp`.
    - **Impact**: The timestamp where the loan was last funded.

Branches and code coverage

Intended branches

- Calculates the right maximum interest amount to pay for the loan.
    - [x] Test coverage

Negative behavior

- Revert if `_lastFundingTimestamp` is bigger than `block.timestamp`.
    - [ ] Negative test
- Revert if `maxInterestToPay` is lower than 0.
    - [ ] Negative test

Function: `computeMaxPrincipal(address, address, uint256 _downPayment)`

This function computes the maximum principal allowed to be borrowed.

Inputs

- `<unnamed@0>`
    - **Impact**: Unused.
- `<unnamed@1>`
    - **Impact**: Unused.
- `_downPayment`
    - **Control**: Full control.
    - **Constraints**: None.
    - **Impact**: The down payment the trader is paying.

Branches and code coverage

Intended branches

- Calculates the right maximum principal allowed to be borrowed.
    - ☐ Test coverage

Negative behavior

- Reverts if `_downPayment` is lower than 0.
    - ☐ Negative test

Function: `constructor(uint256 _maxApy, uint256 _maxLeverage)`

This function allows to initialize the DebtController contract with `_maxApy` and `_maxLeverage`.

Inputs

- `_maxApy`
    - **Control**: None.
    - **Constraints**: None.
    - **Impact**: To DebtController constructor.
- `_maxLeverage`
    - **Control**: None.
    - **Constraints**: None.
    - **Impact**: To DebtController constructor.

5.3. Module: WasabiLongPool.sol

Function: `initialize(IAddressProvider _addressProvider)`

This function initializes the WasabiLongPool contract and assigns the first address provider.

Inputs

- `_addressProvider`
    - **Control**: None.
    - **Constraints**: None.
    - **Impact**: Assigns the first address provider.

Branches and code coverage

Intended branches

- Initializes WasabiLongPool successfully.
    - ☑ Test coverage

Negative behavior

- Reverts if `_addressProvider` is not an AddressProvider contract.
    - ☑ Negative test
- Reverts if `msg.sender` is not the owner.
    - ☑ Negative test
- Reverts if the contract is already initialized.
    - ☐ Negative test

Function call analysis

- `__BaseWasabiPool_init(false, _addressProvider)`
    - **What is controllable?** `_addressProvider`.
    - **If the return value is controllable, how is it used and how can it go wrong?** Not used.
    - **What happens if it reverts, reenters or does other unusual control flow?** N/A.

5.4. Module: WasabiShortPool.sol

Function: `initialize(IAddressProvider _addressProvider)`

This function initializes the WasabiShortPool contract and assigns the first address provider.

Inputs

- `_addressProvider`
    - **Control**: None.
    - **Constraints**: None.
    - **Impact**: Assigns the first address provider.

Branches and code coverage

Intended branches

- Initializes WasabiShortPool successfully.
    - ☑ Test coverage

Negative behavior

- Reverts if `_addressProvider` is not an AddressProvider contract.
    - ☑ Negative test
- Reverts if `msg.sender` is not the owner.
    - ☑ Negative test
- Reverts if hte contract is already initialized.
    - ☐ Negative test

Function call analysis

- `__BaseWasabiPool_init(false, _addressProvider)`
    - **What is controllable?** `_addressProvider`.
    - **If the return value is controllable, how is it used and how can it go wrong?** Not used.
    - **What happens if it reverts, reenters or does other unusual control flow?** N/A.

Function: `openPosition(OpenPositionRequest _request, Signature _signature)`

This function can be used to open a position, long or short, according to the data passed in the `_request` argument.

Inputs

- `_request`
    - **Control**: Unclear; this argument is validated and signed by an off-chain, out-of-scope component.
    - **Constraints**:

- `id` must not be already used.
- `currency` could be anything except a base token.
- `targetCurrency` must be a base token.
- `downPayment` is not directly constrained by the smart contract.
- `principal` must be at most the vault principal balance and is also checked to be no more than the computed maximum principal amount to prevent overleveraging.
- `minTargetAmount` is not directly constrained and is used to implement antislippage checks.
- `expiration` must be after the current `block.timestamp`.
- `fee` is not constrained.
- `functionCallDataList` is not constrained.

- **Impact**: Specifies all the parameters of the position.

- `_signature`
  - **Control**: Arbitrary.
  - **Constraints**: Must be a valid signature of `_request` from the contract owner.
  - **Impact**: Signature authorizing the operation.

Branches and code coverage

Intended branches

- After performing validation of the `_request` argument, it executes the calls specified in `functionCallDataList`, which are intended to grant approval and call Uniswap to perform the needed trade, selling the shorted principal asset for the collateral. After checking the received collateral amount and the principal, the hash identifying the position is recorded in storage.
  - ☑ Test coverage

Negative behavior

- Reverts if the signature is invalid.
  - ☐ Negative test
- Reverts if the position ID is already used.
  - ☐ Negative test
- Reverts if `functionCallDataList` is empty.
  - ☐ Negative test
- Reverts if the request is expired.
  - ☐ Negative test
- Reverts if the `currency` is a base token.
  - ☐ Negative test
- Reverts if the `targetCurrency` is not a base token.
  - ☐ Negative test

- Reverts if receiving the payment fails (for raw ETH).
  - ☐ Negative test
- Reverts if receiving the payment fails (for WETH / other ERC-20).
  - ☐ Negative test
- Reverts if the requested principal is more than the available balance.
  - ☐ Negative test
- Reverts if the received collateral is less than the minumum specified in the request.
  - ☐ Negative test
- Reverts if the requested principal would overleverage the user.
  - ☐ Negative test

Function call analysis

- `this._validateOpenPositionRequest(_request, _signature) -> PerpUtils.receivePayment(this.isLongPool ? _request.currency : _request.targetCurrency, _request.downPayment + _request.fee, this.addressProvider.getWethAddress(), msg.sender)`
  - **What is controllable?** `targetCurrency` and amount (to some extent).
  - **If the return value is controllable, how is it used and how can it go wrong?** Not used.
  - **What happens if it reverts, reenters or does other unusual control flow?** Reentrancy is prevented via `nonReentrant` modifier (apart from admin-only functions).
- `principalToken.balanceOf(address(this))`
  - **What is controllable?** `principalToken` is signed, but it must be validated by the off-chain component.
  - **If the return value is controllable, how is it used and how can it go wrong?** Used as the balance of the principal token before the swap.
  - **What happens if it reverts, reenters or does other unusual control flow?** Reentrancy is prevented via `nonReentrant` modifier (apart from admin-only functions).
- `collateralToken.balanceOf(address(this))`
  - **What is controllable?** `collateralToken` is signed, but it must be validated by the off-chain component.
  - **If the return value is controllable, how is it used and how can it go wrong?** Used as the collateral balance before the swap.
  - **What happens if it reverts, reenters or does other unusual control flow?** Reentrancy is prevented via `nonReentrant` modifier (apart from admin-only functions).
- `PerpUtils.executeFunctions(_request.functionCallDataList)`
  - **What is controllable?** Nothing directly — argument provided by the off-chain component.
  - **If the return value is controllable, how is it used and how can it go wrong?** Not used.

    - **What happens if it reverts, reenters or does other unusual control flow?** Reentrancy is prevented via `nonReentrant` modifier (apart from admin-only functions).
- `collateralToken.balanceOf(address(this))`
    - **What is controllable?** As above, `collateralToken` is signed, but it must be validated by the off-chain component.
    - **If the return value is controllable, how is it used and how can it go wrong?** Used as the collateral balance after the swap.
    - **What happens if it reverts, reenters or does other unusual control flow?** Reentrancy is prevented via `nonReentrant` modifier (apart from admin-only functions).
- `this.addressProvider.getDebtController().computeMaxPrincipal(_request.targetC` `_request.currency, swappedDownPaymentAmount)`
    - **What is controllable?** `targetCurrency` and `currency`, to the extent permitted by the off-chain component. They are currently unused.
    - **If the return value is controllable, how is it used and how can it go wrong?** Used to limit the maximum amount of principal the user can request.
    - **What happens if it reverts, reenters or does other unusual control flow?** The caller cannot cause the call to reenter or revert.
- `principalToken.balanceOf(address(this))`
    - **What is controllable?** `principalToken` is signed and validated by the off-chain component.
    - **If the return value is controllable, how is it used and how can it go wrong?** Used to determine the amount of principal gained from the swap.
    - **What happens if it reverts, reenters or does other unusual control flow?** Reentrancy is prevented via `nonReentrant` modifier (apart from admin-only functions).

5.5. Module: WasabiVault.sol

Function: `depositEth(address receiver)`

This function can be used to deposit ETH into the vault. The ETH is wrapped into WETH, and the receiver is minted a corresponding amount of vault shares.

Inputs

- `receiver`
    - **Control**: Arbitrary.
    - **Constraints**: None.
    - **Impact**: Receiver of the vault shares.

Branches and code coverage

Intended branches

- Wraps the received ETH, mints vault shares to the user, and updates `totalAssetValue`.
  - ☐ Test coverage

Negative behavior

- Reverts if vault asset does not match WETH address returned by address provider.
  - ☐ Negative test
- Reverts if deposited value is zero.
  - ☐ Negative test
- Reverts if deposit amount is too big (cannot happen in practice).
  - ☐ Negative test

Function: `recordInterestEarned(uint256 _interestAmount)`

This function is called by the pool associated with the vault to record incoming interest earned.

Inputs

- `_interestAmount`
  - **Control**: None (only the pool can call this function).
  - **Constraints**: None.
  - **Impact**: Amount of interest earned.

Branches and code coverage

Intended branches

- Increases the recorded `totalAssetValue` by the provided `_interestAmount`.
  - ☐ Test coverage

Negative behavior

- Reverts if the caller is not the pool associated with the vault.
  - ☐ Negative test

Function: `recordLoss(uint256 _amountLost)`

This function is called by the pool associated with the vault to record a loss of assets if a liquidation does not yield sufficient funds.

Inputs

- `_amountLost`
    - **Control**: None (only associated vault can call this function).
    - **Constraints**: None.
    - **Impact**: Amount of lost assets.

Branches and code coverage

Intended branches

- Increases the recorded `totalAssetValue` by the provided `_interestAmount`.
    - ☐ Test coverage

Negative behavior

- Reverts if the caller is not the pool associated with the vault.
    - ☐ Negative test

6. Assessment Results

At the time of our assessment, the reviewed code was deployed to the Ethereum Mainnet.

During our assessment on the scoped Wasabi Perps contracts, we discovered 10 findings. Two critical issues were found. Two were of medium impact, one was of low impact, and the remaining findings were informational in nature. Wasabi 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.