SHERLOCK

Security Review For Wasabi

Collaborative audit prepared for:	**Wasabi**
Lead Security Expert(s):	**EgisSecurity** **0x3b**
Date Audited:	**November 4th - November 11th**

Introduction

Wasabi is a CultureFi DEX. Swap, leverage trade, and stake on-chain culture, starting with meme coins and NFTs.

Scope

Repository: https://github.com/DkodaLabs/wasabi_perps/tree/main

Commit: 23d5b7a4435c1d88e22c6bfc4d5dd0f537aa9394

Findings

Each issue has an assigned severity:

- Medium issues are security vulnerabilities that may not be directly exploitable or may require certain conditions in order to be exploited. All major issues should be addressed.
- High issues are directly exploitable security vulnerabilities that need to be fixed.
- Low/Info issues are non-exploitable, informational findings that do not pose a security risk or impact the system's integrity. These issues are typically cosmetic or related to compliance requirements, and are not considered a priority for remediation.
- General Health issues represent broad areas of concern that may not necessarily be directly related to security vulnerabilities, but can still impact the overall health and performance of the system. These may include areas such as scalability, maintainability, or compliance with industry best practices. While not typically critical in terms of security, addressing General Health issues can help ensure the system remains stable, efficient, and up-to-date with industry standards.

Issues found

High	Medium	Low/Info
0	7	7

Issues not fixed or acknowledged

High	Medium	Low/Info
0	0	0

Issue M-1: User can safe part of the interest repaid if he sandwich his position close with flashloan

Source: https://github.com/sherlock-audit/2024-11-wasabi/issues/62

Summary

When a user opens a position, it starts accruing interest which is paid to vault depositors based on their shares out of the total supply

Vulnerability Detail

There is a possible attack vector where an exploiter can recoup a big portion of the interest repaid. If he uses flashloan of the principal asset to deposit it into the vault, then close his position and repay It afterwards in one transaction, he will manipulate vault shares so he can claim almost the full interest. There are free flash loan providers such as Morpho, where users can borrow millions.

Impact

Almost interest-free positions.

Code Snippet

https://github.com/sherlock-audit/2024-11-wasabi/blob/a13573acccbc1876774bd5f4c50672864b812270/wasabi_perps/contracts/WasabiLongPool.sol#L125-L146

https://github.com/sherlock-audit/2024-11-wasabi/blob/a13573acccbc1876774bd5f4c50672864b812270/wasabi_perps/contracts/vaults/WasabiVault.sol#L16

Tool used

Manual Review

Recommendation

Create deposit/withdraw window on the vault, so user cannot use flash-loan and exploit that.

Discussion

WEBthe3rd

Acknowledged. This can be fixed by not allowing a deposit and withdrawal in the same block. But we want to avoid the extra gas cost to users caused by storing a block number on every deposit, then loading it from storage and comparing it to the current block on every withdrawal. Also, this exploit cannot lead to loss of funds beyond some portion of interest repaid, so we will wait to implement a fix here until we see someone abusing this.

Issue M-2: Blast yield is claimed stepwise, which allows arbitrage of the vault

Source: https://github.com/sherlock-audit/2024-11-wasabi/issues/60

Summary

Blast yield is claimed stepwise, which allows arbitrage of the vault

Vulnerability Detail

Vaults deployed on Blast will use `BlastVault` to claim the yield. However this mechanism introduces step-wise operations, which are instant increases in share value upon claim. This can be arbitraged by MEV bots and users, by simply watching the mempool and sandwiching the `claim` TX.

Example:

1. Alice sees that the admin is gonna claim 1 WETH in 80 WETH pool
2. Alice sandwiches that TX by depositing 20 WETH and then withdrawing it right after the claim
3. Alice hold 20% of the pool shares during the claim operation, so she receives 20% of the value (0.2 ETH)

Impact

Users can MEV `claim`.

Code Snippet

Tool used

Manual Review

Recommendation

Consider adding deposit/withdraw windows.

Discussion

WEBthe3rd

Fixed by adding a `uint256 \_expectedVaultSupply` argument to the claim function, and reverting if `totalSupply() != \_expectedVaultSupply`.

NicolaMirchev

> Fixed by adding a `uint256 \_expectedVaultSupply` argument to the claim function, and reverting if `totalSupply() != \_expectedVaultSupply`.

We have some remaining concerns about the following mitigation. Let's examine the exploit path with the proposed fix:

1. Assume rewards have accrued for 30 days.
2. At this point, totalSupply() = X, and the owner calls claim with _expectedVaultSupply == X.
3. A frontrunner deposits before the claim transaction, causing it to revert.

- From here, two possible scenarios could unfold:
- - There’s a high probability that the admin will attempt to claim again soon after. In this case, the new user may still benefit from a share of the accumulated yield.
- - Alternatively, the admin’s actions may be influenced by the ”attacker,” delaying the claim for another week or month, which could cause dissatisfaction among existing depositors and again inflation of their yield.

In my opinion, the best solution here is to implement a backend bot that automatically claims yield every 2-3 days. This ensures that depositors retain the yield associated with their deposits and don’t inadvertently share it with latecomers.

WEBthe3rd

> In my opinion, the best solution here is to implement a backend bot that automatically claims yield every 2-3 days. This ensures that depositors retain the yield associated with their deposits and don’t inadvertently share it with latecomers.

We actually already have a backend bot for this, which claims yield every 4 hours. So if the claim transaction fails after the frontrunning deposit, the ”attacker” would need to keep their funds in the vault for at least 4 hours to profit from the yield, at which point they would be missing out on the yield they'd receive by just holding the assets themselves, and providing liquidity that could be utilized during that window of time.

NicolaMirchev

> In my opinion, the best solution here is to implement a backend bot that automatically claims yield every 2-3 days. This ensures that depositors retain the yield associated with their deposits and don’t inadvertently share it with latecomers.

> We actually already have a backend bot for this, which claims yield every 4 hours. So if the claim transaction fails after the frontrunning deposit, the ”attacker” would need to keep their funds in the vault for at least 4 hours to profit from the yield, at which point they would be missing out on the yield they'd receive by just holding the assets themselves, and providing liquidity that could be utilized during that window of time.

Okay, great! A little note that blast rebasing happens once every 24 hours.

0x3b33

Fix confirmed

Issue M-3: `claimAllGas` claims gas at a loss

Source: https://github.com/sherlock-audit/2024-11-wasabi/issues/58

Summary

claimAllGas claims less gas than `claimMaxGas`, the difference will be sent to Blast as fees, resulting in a loss for the protocol.

Vulnerability Detail

Blast has a special gas mechanic where contract can claim some percentage of the gas they use. They can do that instantly for 50% of the funds, or wait up to a month in order to linearly unlock 100% of the used gas. In short, if a user uses 1 ETH worth of gas at time T, then at:

1. At T the contract can claim 50%, or 0.5 ETH
2. At T + 2 weeks the contract can 75% or 0.75 ETH
3. At T + 1 month the contract can claim 100% or 1 ETH

All contracts claim their gas using `claimAllGas`, however this function claims 100% of the gas, while paying the fees for the one that is not fully unlocked, i.e. claiming all the gas, even if it' not fully unlocked. This means that every time the admin calls `claimAllGas` the system loses some percentage of all gas that was used inside the contracts for up to 1 month prior.

Example:

1. User uses 1 ETH worth of gas at T
2. User uses 1 ETH worth of gas at T + 1 month
3. Admin calls `claimAllGas`

Now the system will claim the first 1 ETH, but only claim 50% of the second one and send the other 50% to Blast as fees for early claim, resulting in 1.5 ETH instead of 2.

Impact

Less gas will be claimed.

Code Snippet

Tool used

Manual Review

Recommendation

Add a function to call `claimMaxGas` and use it instead. This function will claim all of the gas that is vested at 100% and leave the rest to continue vesting, which will result in claim rate of 100% every time. Blast docs for extra info (if needed).

Discussion

WEBthe3rd

Fixed by changing all calls to `IBlast.claimAllGas` with `claimMaxGas`

0x3b33

Fix confirmed

Issue M-4: `liquidatePosition` differs between the long and short

Source: https://github.com/sherlock-audit/2024-11-wasabi/issues/57

Summary

`liquidatePosition` differs between the long and short

Vulnerability Detail

`liquidatePosition` is differently implemented in `WasabiShortPool` `WasabiLongPool`.

Inside the long pool we calculate `liquidationThreshold` to be 5% of our position `principal`.

https://github.com/sherlock-audit/2024-11-wasabi/blob/main/wasabi_perps/contracts/WasabiLongPool.sol#L156

```
uint256 liquidationThreshold = _position.principal * 5 / 100;
```

However inside the short pool we calculate it as 5% of `collateralAmount`

https://github.com/sherlock-audit/2024-11-wasabi/blob/main/wasabi_perps/contracts/WasabiShortPool.sol#L175

```
uint256 liquidationThreshold = _position.collateralAmount * 5 / 100;
```

where `collateralAmount` is `collateralReceived + _request.downPayment`.

In short that means that for longs `liquidationThreshold` is 5% of what's borrowed and for shorts it's 5% of what's borrowed + 5% of the `downpayment`.

Impact

Short and long liquidation thresholds will be different, where the long will be bigger as it will also include `downpayment`.

Code Snippet

Tool used

Manual Review

Recommendation

This difference will increase the short liquidation threshold to beyond what's borrowed. Consider fixing the discrepancy.

Discussion

WEBthe3rd

Can be fixed by subtracting the down payment from the collateral amount before multiplying by 5%. TBD whether we want to implement this change.

WEBthe3rd

Acknowledged. But changing the liquidation threshold equation would require many changes in our backend liquidation engine, and we don’t think it is worth the risk of unintentionally breaking that.

Issue M-5: WasabiRouter::swapVaultToToken - Function can be DoSed

Source: https://github.com/sherlock-audit/2024-11-wasabi/issues/55

Summary

WasabiRouter::swapVaultToToken - Function can be DoSed

Vulnerability Detail

`swapVaultToToken` performs:

1. Withdraws from a vault in the name of the sender.
2. Either: 2.1 Unwraps the withdrawn WETH from the vault and sends it to `msg.sender` 2.2 Performs a swap between the withdrawn asset and another token. 2.3 Simply transfers the withdrawn asset to `msg.sender`.
3. Before finishing the tx it checks:

```
// SwapRouter should transfer tokenOut directly to user (and fee receiver)
        if (_tokenOut == address(0)) {
            if (address(this).balance != 0) revert InvalidETHReceived();
        } else if (IERC20(_tokenOut).balanceOf(address(this)) != 0) {
            revert InvalidTokensReceived();
        }
```

If either of these is `true` the tx will revert. These checks can very easily be abused and will almost completely brick the function.

```
if (address(this).balance != 0)
```

The contract has a `receive` that can only be called from `weth` and the `swapRouter`, but the contract can be forcefully sent native tokens using `selfdestruct`.

```
else if (IERC20(_tokenOut).balanceOf(address(this)) != 0)
```

Anyone can just transfer 1 wei of `_tokenOut` to the contract.

The only way to currently "fix" this, is by calling `swapTokenToVault` (`swapVaultToVault` might also work), since at the end any tokens either native or ERC20 are refunded back to `msg.sender`.

```
// If full amount of tokenIn was not used, return it to the user
       if (isETHSwap) {
           uint256 amountRemaining = address(this).balance;
           if (amountRemaining != 0) {
               payable(msg.sender).sendValue(amountRemaining);
           }
       } else {
           uint256 amountRemaining = IERC20(_tokenIn).balanceOf(address(this));
           if (amountRemaining != 0) {
               IERC20(_tokenIn).safeTransfer(msg.sender, amountRemaining);
           }
       }
```

This will ”fix” `swapVaultToToken`, but the function can be attacked immediately after.

Impact

Breaking of core functionality, DoS of the function which can easily be repeated

Code Snippet

https://github.com/sherlock-audit/2024-11-wasabi/blob/a13573acccbc1876774bd5f4c50672864b812270/wasabi_perps/contracts/router/WasabiRouter.sol#L165-L169

Tool used

Manual Review

Recommendation

The easiest solution would be to just remove the two checks, but if you want to keep them you cannot revert, as it open up this attack vector.

Discussion

WEBthe3rd

Fixed by removing the `tokenOut` balance checks.

0x3b33

Fix confirmed

Issue M-6: WasabiRouter::swapVaultToToken - No withdraw fee applied when swapping tokens

Source: https://github.com/sherlock-audit/2024-11-wasabi/issues/53

Summary

No withdraw fee applied when swapping tokens

Vulnerability Detail

In 2/3 cases we call `_takeWithdrawFee` which takes a small fee, sends it to the fee receiver and sends the rest to `msg.sender`, but when we call `_swapInternal` it's implied that the swap itself will transfer the swapped tokens to `msg.sender`, skipping the withdraw fee.

```
function swapVaultToToken(
        uint256 _amount,
        address _tokenIn,
        address _tokenOut,
        bytes calldata _swapCalldata
    ) external nonReentrant {
        // Withdraw tokenIn from vault on user's behalf
        _withdrawFromVault(_tokenIn, _amount);

        if (_tokenIn != _tokenOut) {
            if (_tokenOut == address(0) && _tokenIn == address(weth)) {
                // Unwrap WETH to ETH
                weth.withdraw(_amount);
                _takeWithdrawFee(_tokenOut, _amount);
            } else {
                // Perform the swap
                _swapInternal(_tokenIn, _amount, _swapCalldata);
            }
        } else {
            // Transfer the withdrawn assets to user (minus withdraw fee)
            _takeWithdrawFee(_tokenOut, _amount);
        }

        // SwapRouter should transfer tokenOut directly to user (and fee receiver)
        if (_tokenOut == address(0)) {
            if (address(this).balance != 0) revert InvalidETHReceived();
        } else if (IERC20(_tokenOut).balanceOf(address(this)) != 0) {
            revert InvalidTokensReceived();
        }

        // If full amount of tokenIn was not used, return it to the vault
        _depositToVault(_tokenIn, IERC20(_tokenIn).balanceOf(address(this)));
    }
```

Impact

Skipping withdraw fees leading to losses to the protocol

Code Snippet

https://github.com/sherlock-audit/2024-11-wasabi/blob/a13573acccbc1876774bd5f4c50672864b812270/wasabi_perps/contracts/router/WasabiRouter.sol#L155-L158

Tool used

Manual Review

Recommendation

Rework the logic behind `_swapInternal`. Maybe the tokenIn and tokenOut of the swap can be arguments of the function and the rest of the calldata is built around them, this way the logic will be flexible enough to support many different tokens, and you'll still be able to have control over the swap itself and from there you can apply the fee where you see fit.

Discussion

WEBthe3rd

Acknowledged. The `\_takeWithdrawFee` function is only meant to be used for simple withdrawals, where no swaps are performed. For swaps, we intend to use the `SwapRouter02` functions `sweepTokenWithFee` and `unwrapWETH9WithFee`. It is currently possible for advanced users to avoid the swap fee by encoding the `\_swapCalldata` themselves, to make the swap router send the token out directly to them without calling a “with fee” function at the end. But we are okay with this, as the swap fee could be considered a convenience fee for using our app.

Issue M-7: Pools TP orders doesn't prioritize user profit

Source: https://github.com/sherlock-audit/2024-11-wasabi/issues/51

Summary

`WasabiLongPool::closePosition` is used together with user signed order `ClosePositionOrder` to execute a Stop Loss, or Take Profit user orders.

Vulnerability Detail

The problem in `WasabiLongPool` is that we don't prioritize user profit for his TP oder and instead check if the total amount receive from the swap is above user provided `_order.takerAmount`. But the following may result in close order, where user profit is less than a moment in the past, when order was still active:

```
uint256 actualTakerAmount = closeAmounts.payout + closeAmounts.closeFee +
 ↪  closeAmounts.interestPaid + closeAmounts.principalRepaid;

// For Longs, the whole collateral is sold, so the order.takerAmount is the limit
 ↪  amount that the trader expects
// TP: Must receive more than or equal to order.takerAmount
// SL: Must receive less than or equal to order.takerAmount

if (_order.orderType == 0) { // Take Profit
    if (actualTakerAmount < _order.takerAmount) revert PriceTargetNotReached();
} else if (_order.orderType == 1) { // Stop Loss
    if (actualTakerAmount > _order.takerAmount) revert PriceTargetNotReached();
} else {
    revert InvalidOrder();
}
```

Imagine the following scenario:

1. User has opened USDC/WETH long position X4 with 1000 USDC down payment (assume 1 WETH = 2K USDC)
2. User has created TP order with `takerAmount == 4500`
3. WETH price has increased 5% and user PnL is now ($+400 -> position value is worth = 4400 USDC)
4. Some time passes and weth price isn't moving, but interested for user is accrued. Also borrow rate is high -> interest is high

5. After some weeks WETH price increases another 2% and now total position value is worth > 4500 USDC, but user payout would be less than his PnL at 3., because interest has increased at larger scale than weth price.

Impact

NOTE: The problem is not fully fixed in the short pool. There the execution price takes into account the accrued interest, but not the close fee + execution price.

Code Snippet

https://github.com/sherlock-audit/2024-11-wasabi/blob/a13573acccbc1876774bd5f4c50672864b812270/wasabi_perps/contracts/WasabiLongPool.sol#L99-L106

Tool used

Manual Review

Recommendation

Consider using order's `takerAmount` amount as "user profit" and compare it against `closeAmounts.payout`, when we have `_order.orderType == 0` (Take Profit)

Discussion

WEBthe3rd

Acknowledged. We recognize the validity of the issue, and we may introduce an option to create profit-based TP/SL orders in the future, in addition to price-based orders. But currently we don't see demand for this from our users, and would prefer not to add the additional complexity unless there is demand for it.

Issue L-1: User can decrease close fee, if he uses flashloan + claimPosition, instead of `closePosition`

Source: https://github.com/sherlock-audit/2024-11-wasabi/issues/65

Summary

When user uses `claimPosition`, he is charged the `_position.feesToBePaid`, which is the opening fee as close fee:

```
uint256 closeFee = _position.feesToBePaid; // Close fee is the same as open fee
```

But when we use other methods of closing a position (`closePosition`,`liquidate`), we use `PerpUtils.computeCloseFee` to dynamically calculate the closing fee:

```
(closeAmounts.payout, closeAmounts.closeFee) =
    PerpUtils.deduct(
        closeAmounts.payout,
        PerpUtils.computeCloseFee(_position, closeAmounts.payout, isLongPool) +
         ↪   _executionFee);
```

Vulnerability Detail

The thing is that when the user has a positive PnL, `computeCloseFee` will return an amount larger than `_position.feesToBePaid`, so the user can use a flash loan to claim his position and pay a smaller closing fee, claim his position with profit, and repay his flash loan.

Example:

- If user enter with 1e18 weth and borrows 4e18 (leverage X4 with opening fee = 0.02 weth)
- If weth increases 20% and now the user can close his position with 100% profit on his downPayment, closeFee calculation will be 20% above `_position.feesToBePaid` (openFee) =~ 0.024 weth
- User will skip the increased closing fee, if he uses some free flashloan provider to withdraw his profit with a smaller fee, swap it to the flashloan asset and repay it.

Impact

Users bypass real closing fees, if it is beneficial.

Code Snippet

https://github.com/sherlock-audit/2024-11-wasabi/blob/a13573acccbc1876774bd5f4c50672864b812270/wasabi_perps/contracts/WasabiLongPool.sol#L170

Tool used

Manual Review

Recommendation

In `claimPosition` use `computeCloseFee`, instead of using `_position.feesToBePaid`

Discussion

WEBthe3rd

Acknowledged, we will address this down the road if we see anyone abusing the `claimPosition` function.

Issue L-2: Enforce `closeAmounts.collateralSpent == _position.collateralAmount` for long pool

Source: https://github.com/sherlock-audit/2024-11-wasabi/issues/63

Summary

In the long pool, consider checking if we have swapped the whole collateral amount, otherwise the funds stay locked in the contract.

Code Snippet

https://github.com/sherlock-audit/2024-11-wasabi/blob/a13573acccbc1876774bd5f4c50672864b812270/wasabi_perps/contracts/WasabiLongPool.sol#L253-L255

https://github.com/sherlock-audit/2024-11-wasabi/blob/a13573acccbc1876774bd5f4c50672864b812270/wasabi_perps/contracts/WasabiLongPool.sol#L274-L275

Tool used

Manual Review

Recommendation

Consider checking if we have swapped the whole collateral amount

Discussion

WEBthe3rd

Fixed by changing the revert condition in `\_closePositionInternal` from `if(collateralSpent > \_position.collateralAmount)` to `if(collateralSpent != \_position.collateralAmount)`

0x3b33

Fix confirmed

Issue L-3: BaseWasabiPool::_payCloseAmounts - Some tokens revert on 0 value transfers

Source: https://github.com/sherlock-audit/2024-11-wasabi/issues/61

Summary

BaseWasabiPool::_payCloseAmounts - Some tokens revert on 0 value transfers

Vulnerability Detail

In `_payCloseAmounts` the fees are sent to the fee receiver in two ways:

- Native transfer
- ERC20 transfer

The issue with the ERC20 transfer is that, we don't check if `positionFeesToTransfer = 0`, meaning we can attempt a 0 value transfer. There are some ERC20s that revert on 0 value transfers, which will DoS the function, possibly stopping liquidations and normal closing of positions.

```
IERC20 token = IERC20(_token);
token.safeTransfer(_getFeeReceiver(), positionFeesToTransfer);
```

Impact

DoS of the function on rare occasions

Code Snippet

https://github.com/sherlock-audit/2024-11-wasabi/blob/a13573acccbc1876774bd5f4c50672864b812270/wasabi_perps/contracts/BaseWasabiPool.sol#L180-L181

Tool used

Manual Review

Recommendation

Check if `positionFeesToTransfer > 0`, if it's not, just skip the ERC20 transfer.

Discussion

WEBthe3rd

Fixed by checking if `positionFeesToTransfer > 0` and skipping the fee transfer if not. This fee value should always be non-zero, since there is always a fee for opening the position which is paid on close, but we think it makes sense to check anyway just in case we start offering zero-fee trades to users in the future.

0x3b33

Fix confirmed

Issue L-4: BaseWasabiPool::_payCloseAmounts - If `_trader` (the user that owns the position) is both blacklisted from the `token` and/or doesn't accept native tokens, he can DoS liquidations on his positions

Source: https://github.com/sherlock-audit/2024-11-wasabi/issues/59

Summary

BaseWasabiPool::_payCloseAmounts - If `_trader` (the user that owns the position) is both blacklisted from the `token` and/or doesn't accept native tokens, he can DoS liquidations on his positions

Vulnerability Detail

When a position is eligible to be liquidated, the `LIQUIDATOR_ROLE` will call `liquidatePosition`.

Then we enter `_closePositionInternal`, deduct the principal, interest, fees and then if anything is left we will transfer it to the `_trader`, which is done in `_payCloseAmounts`.

```
function _payCloseAmounts(
        PayoutType _payoutType,
        address _token,
        address _trader,
        CloseAmounts memory _closeAmounts
    ) internal {
        uint256 positionFeesToTransfer = _closeAmounts.pastFees +
 ↪   _closeAmounts.closeFee;

        // Check if the payout token is ETH/WETH or another ERC20 token
        address wethAddress = _getWethAddress();
        if (_token == wethAddress) {
            uint256 total = _closeAmounts.payout + positionFeesToTransfer +
 ↪   _closeAmounts.liquidationFee;
            IWETH wethToken = IWETH(wethAddress);
            if (_payoutType == PayoutType.UNWRAPPED) {
                if (total > address(this).balance) {
                    wethToken.withdraw(total - address(this).balance);
                }
                PerpUtils.payETH(positionFeesToTransfer, _getFeeReceiver());

                if (_closeAmounts.liquidationFee > 0) {
                    PerpUtils.payETH(_closeAmounts.liquidationFee,
↪   _getLiquidationFeeReceiver());
                }

                PerpUtils.payETH(_closeAmounts.payout, _trader);
                // Do NOT fall through to ERC20 transfer
                return;
            } else {
                uint256 balance = wethToken.balanceOf(address(this));
                if (total > balance) {
                    wethToken.deposit{value: total - balance}();
                }
                // Fall through to ERC20 transfer
            }
        }

        IERC20 token = IERC20(_token);
        token.safeTransfer(_getFeeReceiver(), positionFeesToTransfer);

        if (_closeAmounts.liquidationFee > 0) {
            token.safeTransfer(_getLiquidationFeeReceiver(),
↪   _closeAmounts.liquidationFee);
        }

        if (_closeAmounts.payout != 0) {
            if (_payoutType == PayoutType.VAULT_DEPOSIT) {
                IWasabiVault vault = getVault(address(token));
                if (token.allowance(address(this), address(vault)) <
↪   _closeAmounts.payout) {
                    token.approve(address(vault), type(uint256).max);
                }
                vault.deposit(_closeAmounts.payout, _trader);
            } else {
                token.safeTransfer(_trader, _closeAmounts.payout);
            }
        }
    }
```

Here we have several options:

1. Transfer the funds as native
2. Transfer the funds as ERC20s
3. Transfer the funds as ERC20s and the `closeAmount.payout` which is the `trader` part, gets deposited into a vault.

The issue here is that 1 & 2 can be DoSed by the `trader`.

1. Can be DoSed if the `trader` doesn't accept native tokens or simply reverts in his `fallback/receive` functions.
2. Can be DoSed if the `trader` is blacklisted from the token, this way any transfers from & to him will always revert.

There is an exception to point 2, if `closeAmount.payout = 0` then we don't attempt to transfer any tokens. This is a very unlikely scenario as this likely means that the trader's position is insolvent, since he can't cover his loan + fees, so it's a rare scenario.

Impact

2 out of the 3 paths can be DoSed, which increases the chances of insolvency for a position

Code Snippet

https://github.com/sherlock-audit/2024-11-wasabi/blob/a13573acccbc1876774bd5f4c50672864b812270/wasabi_perps/contracts/BaseWasabiPool.sol#L169
https://github.com/sherlock-audit/2024-11-wasabi/blob/a13573acccbc1876774bd5f4c50672864b812270/wasabi_perps/contracts/BaseWasabiPool.sol#L195

Tool used

Manual Review

Recommendation

Wrapping the two external calls in a try/catch is an option. Another option would be to add an extra flag that the liquidator can pass, which if set to `true`, doesn't send the payout to the `trader`, but to some other protocol address. Considering liquidators are trusted actors, this might be a better option, since using try/catch in Solidity can lead to more issues.

Discussion

WEBthe3rd

Partially fixed by ignoring the `PayoutType.UNWRAPPED` flag if `\_trader.code.length > 0`, such that if the payout token is WETH and the recipient is a contract, they will always receive WETH or, if `\_payoutType == PayoutType.VAULT\_DEPOSIT`, WETH vault shares.

We acknowledge the blacklisted trader issue, but implementing a fix is tricky because tokens containing blacklists are non-standard, and checking if an address is blacklisted can vary between implementations. We agree that using a try/catch could cause other issues, but rather than adding an extra flag that causes payouts to be sent to some other address, we can use the `PayoutType.VAULT\_DEPOSIT` flag as a fallback for liquidations.

WEBthe3rd

Fyi, we have decided to remove the `\_trader.code.length` check, and instead just always use `PayoutType.VAULT\_DEPOSIT` for liquidations.

Issue L-5: `getVault` is made to work with `address(0)`, but everything else is not

Source: https://github.com/sherlock-audit/2024-11-wasabi/issues/56

Summary

`getVault` is made to work with `address(0)`, but everything else is not

Vulnerability Detail

getVault is made to work with `address(0)`

https://github.com/sherlock-audit/2024-11-wasabi/blob/main/wasabi_perps/contracts/BaseWasabiPool.sol#L99-L105

However the rest of the codebase will revert if `address(0)` is provided instead of `wethAddress`. Example is `openPositionFor`, where if `_request.currency` is `address(0)` the vault will be WETH, but the rest of the function will not work and revert.

https://github.com/sherlock-audit/2024-11-wasabi/blob/main/wasabi_perps/contracts/WasabiShortPool.sol#L39-L50

Impact

Useless code, takes up gas and increases contract size.

Code Snippet

Tool used

Manual Review

Recommendation

Remove the code to lower contract size and save gas, or optimize the rest of the codebase to count `address(0)` as WETH.

Discussion

WEBthe3rd

Addressed by removing the `address(0)` handling from `getVault`.

0x3b33

Fix confirmed

Issue L-6: BlastRouter enables WETH/USDB yield, but cannot claim it

Source: https://github.com/sherlock-audit/2024-11-wasabi/issues/54

Summary

`BlastRouter` inherits from `AbstractBlastContract`, which configures yield mode of WETH and USDB on claimable:

```
IERC20Rebasing(BlastConstants.USDB).configure(YieldMode.CLAIMABLE);
IERC20Rebasing(BlastConstants.WETH).configure(YieldMode.CLAIMABLE);
```

Although `BlastRouter` is designed to not hold any funds, there may be some left/transferred WETH/USDB, which yield won't be claimable

Impact

Unclaimable yield

Code Snippet

https://github.com/sherlock-audit/2024-11-wasabi/blob/a13573acccbc1876774bd5f4c50672864b812270/wasabi_perps/contracts/vaults/BlastVault.sol#L25

Tool used

Manual Review

Recommendation

Consider adding function to claim rebasing tokens yield

Discussion

WEBthe3rd

Acknowledged. The router is designed to not hold any WETH/USDB though, so we don't think it makes sense to add claimYield functionality to this contract. Since the contract is upgradeable, if it is ever the case that some rebasing tokens are leftover in the router which accrue enough yield to make claiming it worthwhile, we can add a claim function then.

Issue L-7: `setMaxDailyAPY` should be `setMaxYearlyAPY`, because the APY is based on one year

Source: https://github.com/sherlock-audit/2024-11-wasabi/issues/52

Summary

Change `setMaxDailyAPY` func name, because APY % is based on one year, instead of a day:

```
function computeMaxPrincipal(
    address,
    address,
    uint256 _downPayment
) external view returns (uint256 maxPrincipal) {
    maxPrincipal = _downPayment * (maxLeverage - LEVERAGE_DENOMINATOR) /
     ↪  LEVERAGE_DENOMINATOR;
}
```

Impact

QA/Low

Code Snippet

https://github.com/sherlock-audit/2024-11-wasabi/blob/a13573acccbc1876774bd5f4c50672864b812270/wasabi_perps/contracts/debt/DebtController.sol#L57

Tool used

Manual Review

Discussion

WEBthe3rd

Fixed by renaming the function to `setMaxAPY`, since the A in APY already implies an annual rate.

0x3b33

Fix confirmed

Disclaimers

Sherlock does not provide guarantees nor warranties relating to the security of the project.

Usage of all smart contract software is at the respective users’ sole risk and is the users’ responsibility.