Obsidian

AUDITS

Oku Trade Security Review

Auditors

0xjuaan

0xSpearmint

23rd July, 2025

Introduction

Obsidian Audits

Obsidian audits is a team of top-ranked security researchers, with a publicly proven track record, specialising in DeFi protocols across EVM chains and Solana.

The team has achieved 10+ top-2 placements in audit competitions, placing #1 in competitions for Yearn Finance, Pump.fun, and many more.

Find out more: obsidianaudits.com

Audit Overview

Oku Trade (https://x.com/okutrade) is a non-custodial DeFi meta-aggregator offering the best swap and bridge rates across any EVM chain. Create market and limit orders, manage Uniswap v3 liquidity positions, and on/off ramp directly to any EU or US bank with zero fees.

Oku Trade engaged Obsidian Audits to perform a security review on their custom order type smart contracts. The 5-day review took place from the 16th to the 20th of July, 2025.

Scope

Files in scope

Repo	Files in scope
`automatedTrigger` **Commit hash:** fbd5458	contracts/automatedTrigger/*.sol

Fix Review commit: e5e38a1

Summary of Findings

Severity Breakdown

A total of **12** issues were identified and categorized based on severity:

- **1 High severity**
- **1 Medium severity**
- **3 Low severity**
- **7 Informational**

Findings Overview

ID	Title	Severity	Status
H-01	Anyone can fill a bracket order with zero-amount swaps, effectively cancelling it	High	Fixed
M-01	Creating orders with permit signatures can be frontran and DOS'd	Medium	Fixed
L-01	Fees are deducted after swap execution and slippage checks, allowing true output to be less than `amountOutMin`	Low	Fixed
L-02	The owner can steal or lock user funds through multiple methods	Low	Acknowledged
L-03	Slippage check can fail due to precision loss in `adjustForDecimals()`	Low	Fixed
I-01	A compromised admin can steal funds from the order contracts through the arbitrary call in `execute()`	Informational	Acknowledged
I-02	Duplicate check in `OracleLess::modifyOrder()`	Informational	Fixed
I-03	Unnecessary parameter in `AutomationMaster.pauseAll()`	Informational	Fixed

ID	Title	Severity	Status
I-04	Malicious keepers can fill orders while stealing the slippage amount	Informational	Acknowledged
I-05	`generateOrderId()` is callable by any address	Informational	Fixed
I-05	Unnecessary zero-approval of tokens	Informational	Fixed
I-06	Tokens that revert on 0 approval are unusable with the contracts	Informational	Acknowledged

Fix review Findings

ID	Title	Severity	Status
C-01	Contracts can be drained due to a lack of token validation during `permitTransferFrom()`	Critical	Fixed

Severity Classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Info

Impact

- **High -** leads to a significant material loss of assets in the protocol or significantly harms a group of users.
- **Medium -** leads to a moderate material loss of assets in the protocol or moderately harms a group of users. Alternatively, breaking a core aspect of the protocol's intended functionality
- **Low -** leads to a minor material loss of assets in the protocol or harms a small group of users.

Likelihood

- **High -** attack path is possible with reasonable assumptions that mimic on-chain conditions, and the cost of the attack is relatively low compared to the amount of funds that can be stolen or lost.
- **Medium -** the attack/vulnerability requires minimal or no preconditions, but there is limited or no incentive to exploit it in practice
- **Low -** requires highly unlikely precondition states, or requires a significant attacker capital with little or no incentive.

Findings

[H-01] Anyone can fill a bracket order with zero-amount swaps, effectively cancelling it

Description

The protocol is designed to allow users to create bracket orders, which are filled by keepers once a price condition is met. These orders are intended to be fully executed when either a stop-loss or take-profit threshold is triggered, ensuring that users can exit positions at defined price levels.

However, the keeper can submit `txData` that swaps zero tokens (or a negligible amount like 100 wei). The contract treats this as a valid fill, removes the order from the `dataSet`, and refunds the remaining `tokenIn` to the user. There is no enforcement that the entire `amountIn` is used during execution.

This has a severe impact on traders that trusted the system for their stop-losses / take-profit orders, effective cancellation of their order leaves them exposed to further market movements.

Attack path

- Current ETH market price = $3.5k
- User creates an order with the following params:
  - amountIn = 10 ETH
  - takeProfit = $5k
- Once ETH reaches $5k, a malicious keeper calls `performUpkeep()`, performing a swap on a swap router. The `txData` of the swap specifies `amountIn = 100 wei`
  - The negligible swap executes, closing the order, and the remaining `10 ether - 100 wei` is returned to the order creator
- ETH price crashes to $4k

Since the user's order did not execute, they were still exposed to ETH and were caused to lose $10k from the peak (10 * (5k - 4k)).

Recommendation

One solution would be to allow users to specify a `minAmountOut` threshold, and check that the keeper fulfills that `minAmountOut` in their order to prevent tiny fills.

Alternatively, if partial fills are required, then consider not deleting an order until it is fully filled (amountIn is entirely consumed).

Remediation: Fixed in commit e7885cb

[M-01] Creating orders with permit signatures can be frontran and DOS'd

Description

Whenever creating or modifying orders, `procureTokens()` is used to obtain tokens from the user. They can pass in `permit=true` and a `permitPayload` if they intend to use a permit2 approval rather than a normal erc20 approval to the order contract.

```
if (permit) {
    require(amount < type(uint160).max, "uint160 overflow");
    IAutomation.Permit2Payload memory payload = abi.decode(
        permitPayload,
        (IAutomation.Permit2Payload)
    );

    permit2.permit(tokenOwner, payload.permitSingle, payload.signature);
    permit2.transferFrom(
        tokenOwner,
        address(this),
        uint160(amount),
        address(token)
    );
}
```

The issue is that a malicious actor can observe a user's `permitPayload` in the mempool and frontrun their transaction, calling `permit2.permit()` directly, with the same parameters. This will invalidate the user's nonce in Permit2, causing their order creation transaction to revert.

Recommendation

Consider reading from the `allowance` mapping in the Permit2 contract, to check if the order contract already has sufficient approval from the user, via permit2. If so, ignore the `permit2.permit()` call.

Remediation: Fix introduced a critical vulnerability, explained here

[L-01] Fees are deducted after swap execution and slippage checks, allowing true output to be less than `amountOutMin`

Description

When filling an order, `execute()` is called to perform the swap, and then `applyFee()` is called to deduct the fee from the swap's output amount.

The `minAmountOut` slippage check occurs during `execute()`, before fees are applied. This means that the true output amount to the user can be less than the `minAmountOut`, due to the fee being deducted after the slippage constraint.

Example

1 WETH = 3000 USDC
amountIn = 1 WETH
amountOutMin = 2970 USDC

`execute()` swaps 1 WETH for 2970 USDC, slippage check passes
`applyFee()` then charges a 0.5% fee, leaving 2955.15 USDC for the user, which is less than `amountOutMin`

Recommendation

One solution would be to increase `amountOutMin` by the fee amount (dividing by 1 - fee)

For example, the slippage constraint should check that the swap returned at least 2970/(1 - 0.005) ~= 2984.92 USDC
This would ensure that after applying the 0.5% fee, the minimum output to the user is at least $2970 USDC, as specified by `amountOutMin`

Remediation: Fixed in commit 8a5104c

[L-02] The owner can steal or lock user funds through multiple methods

Description

While it's documented that the contract's owner should not be able to steal user funds, there are multiple vectors that make this possible:

1. Set a malicious target setter, who can register malicious swap targets that can be used to drain the contract of all tokens, and steal all approved tokens (the attack is described in issue I-01)
2. Configure 100% fees via the front-end when creating orders- so when the order is filled, all the proceeds will go to the protocol.

Recommendation

1. the recommended fix is detailed in issue I-01
2. Consider enforcing an upper limit on `feeBips` during order creation. A hard cap of 500 bps (5%) prevents full drain scenarios from a malicious/compromised front end.

Remediation: Issue acknowledged

[L-03] Slippage check can fail due to precision loss in `adjustForDecimals()`

Description

The `execute()` function in the `Bracket` contract performs a slippage check by calling `AutomationMaster.getMinAmountReceived()`

The `getMinAmountReceived()` function calculates the minimum amount out by using the current exchange rate to convert the input to output, then adjusting that output by the slippage %.

It firsts adjusts the input token's decimals to match the output using `adjustForDecimals()` function.

An issue can occur when swapping from tokens of higher decimals to lower, as the following calculation in `adjustForDecimals()` function can encounter significant precision loss:

```
if (decimalIn > decimalOut) {
    // Reduce amountIn to match the lower decimals of tokenOut
    return amountIn / (10 ** (decimalIn - decimalOut));
```

The issue is most pronounced when working with low-decimal output tokens like GUSD (although GUSD is not currently on EVM L2s).

Consider the following example, when swapping from WBTC (8 decimals) into GUSD (2 decimals):

1. User wants to swap 9e5 WBTC (worth $900 USD assuming 1 BTC = $100k) into GUSD
2. When performing the slippage check, `adjustForDecimals()` will adjust the `amountIn` to 0 (due to dividing by `1e6`), as a result any output amount of the swap will be considered valid and pass the slippage check (even 0 output).

This exposes users to high amounts of value extraction from keepers, as the slippage check is basically useless in this scenario.

Recommendation

The root cause of this issue is that division in `adjustForDecimals()` occurs before multiplication when calculating `fairAmountOut`

The solution would be to ensure that if `adjustForDecimals()` is performing division, the division is performed after the multiplication when calculating `fairAmountOut`, not before.

Example

Currently, the calculation is `fairAmountOut = (9e5 / 1e6) * 100000e8 / 1e8` = `0 * 100000` = `0 GUSD`

To avoid precision loss, division by `1e6` should be moved to the end: `9e5 * 100000e8 / 1e8 / 1e6` = `9e5 GUSD` = `$900`

Remediation: Fixed in commit e5e8d2a

[I-01] A compromised admin can steal funds from the order contracts through the arbitrary call in `execute()`

Description

Both the `Bracket` and the `OracleLess` order contracts will execute a swap via `execute()` to fill the order.

The `execute()` function performs the swap by executing a call to the `target` address with `txData`, both which are passed in by the caller.

```
(bool success, bytes memory result) = target.call(txData);
```

The issue is that an attacker can pass in a token as the `target`, and pass a crafted `txData` that will approve the contract's token balance to the attacker. After that call, the attacker will transfer these approved funds from the contract to themselves.
The attacker can also pass a `txData` which will call `transferFrom()` on the token to steal any funds that are approved to the contract.

The attacker can ensure the `performUpkeep()` call does not fail due to 0 `amountOut` from the swap, by previously creating an order with 100% slippage tolerance.

Attack path

1. Create an order in `OrderLess` or `Bracket`, with tokenA as input, tokenB as output.
2. Fulfill the order with `target=tokenC` and `txData=abi.encodeWithSelector(IERC20.approve.selector, attacker, tokenCBalance)`
3. Call `tokenC.transferFrom(address(orderContract), attacker, approvedAmount)` to steal the entire balance of the contract
   Alternatively, step 2 can use `abi.encodeWithSelector(IERC20.transferFrom.selector, victim, attacker, amount)` where `victim` is a user with approved funds to the order contract

Note: Currently the `target` must be a whitelisted address, but the audit scope was still interested in such issues as a primary concern:

> The intent is to place zero trust in the data returned by off chain automation, such that the contracts control all aspects of security. This is the primary concern: Is there any ability for mailicious target or txData to be used to compromise user funds? This would be a critical vulnerability.

Recommendation

One solution would be to ensure that upon `execute()`, at least 1 wei of the output token is received, and that 1 wei of the input token is spent. This ensures that a swap must have occured, and the transaction will revert if a token was used as the `target`.

Alternatively, rather than making the arbitrary call from the `OracleLess` or `Bracket` contract, consider using a separate `Executor` contract which performs the arbitrary calls to swap routers after tokens are sent to it during order fulfillment. The `Executor` should only hold funds transiently, and hold no user approvals, so malicious `target` or `txData` values will have no impact on the protocol.

Remediation: Issue acknowledged

[I-02] Duplicate check in `OracleLess::modifyOrder()`

Description

The `OracleLess::modifyOrder()` function includes the following check:

```
require(dataSet.contains(orderId), "order not active");
```

Then, in the internal `_modifyOrder()` function, the same check is unnecessarily used again, as seen here.

Recommendation

Since the same check appears twice, it can be removed in one of the instances

Remediation: Fixed in commit 95dfbe4

[I-03] Unnecessary parameter in `AutomationMaster.pauseAll()`

Description

`pauseAll()` currently accepts `oracleLessContract` as a parameter, in order to pause it.

```
function pauseAll(
    bool pause,
    IOracleLess oracleLessContract
) external override onlyOwner {
```

However, the `OracleLess` contract is already stored as a storage variable, which can be used instead:

```
IOracleLess public ORACLELESS_CONTRACT;
```

Recommendation

Consider removing the parameter and using the stored variable instead

Remediation: Fixed in commit 15b1462

[I-04] Malicious keepers can fill orders while stealing the slippage amount

Description

Since orders can be executed by anyone, a malicious keeper can use a flash loan to atomically manipulate the pool price when executing `Bracket` orders, extracting as much value as slippage allows.

Here is an example:

1. User creates order:

- amountIn = 1 wBTC, stopPrice = $100k, slippage = 5% (current price = $120k)

2. Market price hits $100k.

3. The keeper first uses flash loaned funds to sell WBTC in the DEX pool to manipulate the WBTC price to $95k. Then they execute `performUpkeep()`, filling the order with only 95,000 USDC, this will pass the 5% slippage threshold.
4. Then the keeper buys back the sold WBTC in step 3 to revert the pool to the original price, they profit the 5,000 USDC minus any fees paid.

Recommendation

One way to mitigate against this would be to whitelist trusted keepers.
Alternatively, to limit potential harm, consider adding a constraint to ensure a maximum slippage tolerance.

Remediation: Issue acknowledged

[I-05] `generateOrderId()` is callable by any address

Description

The `generateOrderId()` function is publicly callable by any address. While intended to only be used by sub keeper contracts, it currently lacks access control. This violates the security principle of least privilege, as untrusted callers have access to state changing functions unnecessarily.

Recommendation

Consider restricting the `generateOrderId()` function to only be callable by the designated sub keeper contracts.

An example modifier:

```
modifier onlySubKeepers() {
    require(
        msg.sender == address(BRACKET_CONTRACT) ||
        msg.sender == address(STOP_LIMIT_CONTRACT) ||
        msg.sender == address(ORACLELESS_CONTRACT),
        "Not authorized"
    );
    _;
}
```

Remediation: Fixed in commit d369c72

[I-05] Unnecessary zero-approval of tokens

Description

When input tokens are approved for a swap, the contracts use the following flow:

1. token.decreaseAllowance(existingAllowance)
2. token.increaseAllowance(amountIn)
3. Execute swap
4. token.decreaseAllowance(remainingAllowance)

This is done because some tokens like USDT require that the existing allowance is 0 in order to set a non-zezro allowance.

The first step in the above sequence is redundant, because the last step already reduces approvals to zero, so there is no need to do it again in the first step.

Recommendation

Consider removing the initial zero-approval in `Bracket`, `OracleLess`, and `StopLimit`

Instances:

- Bracket.sol#L616-620
- OracleLess.sol#L346-350
- StopLimit.sol#L152-156

Remediation: Fixed in commit e5e38a1

[I-06] Tokens that revert on 0 approval are unusable with the contracts

Description

Certain tokens like BNB will revert when attempting to approve a 0 amount.

In the `execute()` function, it first attempts to set the approval to 0, therefore for tokens like BNB `execute()` will revert, therefore orders involving such tokens cannot be fulfilled.

```
//approve 0
        tokenIn.safeDecreaseAllowance(
            target,
            (tokenIn.allowance(address(this), target))
        );
```

Note: The protocol is intended to be deployed on L2's and no major tokens on L2's currently exhibit this behaviour. The issue was submitted to document the edge case, in case there is a future token that exhibits such behaviour that the protocol may want to integrate.

Remediation: Issue acknowledged

Fix Review

[C-01] Contracts can be drained due to a lack of token validation during `permitTransferFrom()`

Description

Rather than the recommended fix, the decision was to use `SignatureTransfer` from permit2 to mitigate against M-01 (which is also a valid mitigation). The contracts use `permitTransferFrom()` to obtain funds from the user:

```
IAutomation.Permit2Payload memory payload = abi.decode(
    permitPayload,
    (IAutomation.Permit2Payload)
);
require(payload.permitTransferFrom.permitted.token == token);

IPermit2.SignatureTransferDetails memory transferDetails =
IPermit2.SignatureTransferDetails({
    to: address(this),
    requestedAmount: amount
});

permit2.permitTransferFrom(
    payload.permitTransferFrom,
    transferDetails,
    tokenOwner,
    payload.signature
```

```
);
```

The signed payload (`payload.permitTransferFrom`) contains the following information:

```
/// @notice The token and amount details for a transfer signed in the
permit transfer signature
struct TokenPermissions {
    // ERC20 token address
    address token;
    // the maximum amount that can be spent
    uint256 amount;
}

/// @notice The signed permit message for a single token transfer
struct PermitTransferFrom {
    TokenPermissions permitted;
    // a unique value for every token owner's signature to prevent
signature replays
    uint256 nonce;
    // deadline on the permit signature
    uint256 deadline;
}
```

The issue is that there is no validation that the `token` is the same as `order.tokenIn`.

This allows a malicious actor to perform the following attack:

1. Create a new ERC20 and mint the supply to the attacker's address. Approve it to permit2.
2. Create an order with `amountIn=tokenIn.balanceOf(orderContract)`.
3. In the order, pass in a permit payload with `TokenPermissions.token=attackerToken`
4. The order creation will succeed, as the contract procures the user's worthless tokens without validation
5. The attacker can call `cancelOrder()` to close the order and refund `amountIn` of `order.tokenIn`, draining the contract
6. The attack can be repeated atomically to drain all tokens in the contract

Recommendation

Add a check to ensure that the token being transferred is the same as the `token` passed in to `procureTokens()`

```
IAutomation.Permit2Payload memory payload = abi.decode(
    permitPayload,
    (IAutomation.Permit2Payload)
);
require(payload.permitTransferFrom.permitted.token == token);
```

Remediation: Fixed in commit 15baf61