yAudit Sickle Update Review

Review Resources:

The code repositories and a gist of things that had changed since the last review.

Auditors:

- HHK
- Jackson

Table of Contents

Review Summary

Sickle

This review is an update to a prior review of the Sickle contracts done in June 2023. Only the new updates were in scope. These new updates include a simplified strategy architecture, leading to the deprecation of the `PermitManager`, `FlashloanOpsRegistry`, and `CashManager` contracts, and the replacement of the `FlashloanStub` with a `FlashloanStrategy`.

The contracts of the Sickle Protocol were reviewed over 7 days. The code review was performed by 2 auditors between October 2, 2023, and October 9, 2023. The repository was not under active development during the review, and the review was limited to the latest commit at the start of the review. This was commit 8e0e9d5fcb328857182eb948fa682f81d1474913 for the Sickle repo.

Scope

The scope of the review consisted of the following contracts at the specific commit:

```
contracts
├── Sickle.sol
├── SickleFactory.sol
├── SickleRegistry.sol
├── base
│   ├── Admin.sol
│   ├── Multicall.sol
│   └── SickleStorage.sol
├── libs
│   └── FeesLib.sol
└── strategies
    ├── AerodromeStrategy.sol
    ├── DForceStrategyOne.sol
    ├── DForceStrategyTwo.sol
    ├── FlashloanStrategy.sol
    ├── MasterChefStrategy.sol
    └── StrategyBase.sol
```

After the findings were presented to the Sickle team, fixes were made and included in several PRs.

This review is a code review to identify potential vulnerabilities in the code. The reviewers did not investigate security practices or operational security and assumed that privileged accounts could be trusted. The reviewers did not evaluate the security of the code relative to a standard or specification. The review may not have identified all potential attack vectors or areas of vulnerability.

yAudit and the auditors make no warranties regarding the security of the code and do not warrant that the code is free from defects. yAudit and the auditors do not represent nor imply to third parties that the code has been audited nor that the code is free from defects. By deploying or using the code, Sickle and users of the contracts agree to use the code at their own risk.

Code Evaluation Matrix

Category	Mark	Description
Access Control	Good	The appropriate functions are access-controlled with the appropriate actors.
Mathematics	Good	The mathematics are simple and secured by solidity 0.8 making overflow and underflow revert.
Complexity	Average	The architecture is simple but some elements that could be off-chain are processed on-chain increasing the overall complexity. Additionally, the update introduces proxy and extensive use of `delegateCall`.
Libraries	Good	The libraries used are appropriate and used correctly.
Decentralization	Average	There are some trusted actors that can introduce attack vectors.
Code stability	Good	The repository was not under active development during the review.
Documentation	Average	The contracts are documented through NatSpec comments.
Monitoring	Low	Strategies don't emit events.

Category	Mark	Description
Testing and verification	Good	Lots of effort was put into testing since the previous audit. Almost 100% test coverage is achieved on the codebase with some fuzzing and onchain fork.

Findings Explanation

Findings are broken down into sections by their respective impact:

- Critical, High, Medium, Low Impact
  - These are findings that range from attacks that may cause loss of funds, impact control/ownership of the contracts, or cause any unintended consequences/actions that are outside the scope of the requirements
- Gas savings
  - Findings that can improve the gas efficiency of the contracts
- Informational
  - Findings including recommendations and best practices

Critical Findings

None.

High Findings

None.

Medium Findings

1. Medium - Calling `dForce_claimDFTokenRewards()` will not charge the correct fee

In the strategy DForceStrategyOne.sol the fee applied when calling directly the function `dForce_claimDFTokenRewards()` will not be the correct fee as it will always charge the `dForce_compoundRewardsWithFlashloan` fee tier.

Technical Details

In the function `dForce_claimDFTokenRewards()` the fee applied in the callback `DFORCE_CLAIMDFTOKENREWARDS()` will always be `dForce_compoundRewardsWithFlashloan` and not `dForce_claimDFTokenRewards`.

This is because the callback is called through a `delegatecall()` from the Sickle which doesn't share the same storage. Thus, when checking `isInternalCall` the contract will read the first storage slot of the Sickle contract which will be `_initialized` and `_initializing` from the `Initializable` contract that will result in always returning `true`.

Here is a POC showing that the wrong fee is charged (can be copied and pasted in DForceStrategy.t.sol):

```
function test_badFee(
        uint256 supplyAmount,
        uint256 tokenArrayIndex,
        uint256 leverageLevelInPercentage,
        uint256 flashloanProviderArrayIndex,
        uint256 rewardAccumulatingPeriodInBlocks
    ) public prank {
        // picking a stablecoin at random
        tokenArrayIndex = bound(tokenArrayIndex, 0, 2);
        // minimum of 1000 tokens and maximum 100 000 tokens supplied
        supplyAmount = bound(
            supplyAmount,
            1000
                * 10
                    ** IERC20Metadata(suppliedTokensArray[tokenArrayIndex]).decimals(),
            100_000
                * 10
                    ** IERC20Metadata(suppliedTokensArray[tokenArrayIndex]).decimals()
        );
        // leverage between 101% and 550%
        leverageLevelInPercentage = bound(leverageLevelInPercentage, 101, 550);
        // picking a flashloan provider at random
        flashloanProviderArrayIndex = bound(flashloanProviderArrayIndex, 0, 4);
        // ensuring a minimum deposit time of one day so rewards can be
        // accumulated
        rewardAccumulatingPeriodInBlocks = bound(
            rewardAccumulatingPeriodInBlocks,
            (60 * 60 * 24) / 12,
            (60 * 60 * 24 * 3) / 12
        );

        // establish a folding position on dForce
        vm.stopPrank();
        address[] memory targetCollateralAssetArray =
        test_leverageWithFlashloan_should_succeed(
```

```
        supplyAmount,
        tokenArrayIndex,
        leverageLevelInPercentage,
        flashloanProviderArrayIndex
    );
    vm.startPrank(sickleOwner);

    // fast-forwarding the chain to accumulate rewards
    vm.roll(block.number + rewardAccumulatingPeriodInBlocks);

    // update rewards
    {
        address[] memory holders = new address[](1);
        holders[0] = address(ctx.sickle);
        address[] memory iTokens = new address[](1);
        iTokens[0] = iTokensArray[tokenArrayIndex];
        IDForceRewardDistributor(Mainnet.DFORCE_REWARD_DISTRIBUTOR)
            .updateRewardBatch(holders, iTokens);

        uint256 previousRewardsBalance =
            IERC20(Mainnet.DFORCE_DF).balanceOf(sickleOwner);

        //calculate rewards that should be received with the right fee
        uint256 goodRewardAmountAfterFee = IDForceRewardDistributor(
            Mainnet.DFORCE_REWARD_DISTRIBUTOR
        ).reward(address(ctx.sickle))
            * (
                10_000
                    - SickleRegistry(ctx.registry).feeRegistry(
                        keccak256(
                            abi.encodePacked(
                                address(dForceStrategyOne),
                                dForceStrategyOne
                                    .dForce_claimDFTokenRewards
                                    .selector
                            )
```

```
                )
            )
    ) / 10_000;

//calculate rewards that should be received with the wrong fee
uint256 wrongRewardAmountAfterFee = IDForceRewardDistributor(
    Mainnet.DFORCE_REWARD_DISTRIBUTOR
).reward(address(ctx.sickle))
    * (
        10_000
            - SickleRegistry(ctx.registry).feeRegistry(
                keccak256(
                    abi.encodePacked(
                        address(dForceStrategyOne),
                        dForceStrategyOne
                            .dForce_compoundRewardsWithFlashloan
                            .selector
                    )
                )
            )
    ) / 10_000;

//claim rewards
dForceStrategyOne.dForce_claimDFTokenRewards(
    address(ctx.sickle), iTokens, iTokens, true
);

//rewards increased
assertGt(
    IERC20(Mainnet.DFORCE_DF).balanceOf(sickleOwner),
    previousRewardsBalance
);
//rewards increased by the wrong amount since we didn't charge the
// right fee
assertLt(
    IERC20(Mainnet.DFORCE_DF).balanceOf(sickleOwner),
```

```
                previousRewardsBalance + goodRewardAmountAfterFee
            );
            assertEq(
                IERC20(Mainnet.DFORCE_DF).balanceOf(sickleOwner),
                previousRewardsBalance + wrongRewardAmountAfterFee
            );
        }
    }
```

Impact

Medium. Users may be charged extra fees.

Recommendation

Call the strategy view function `DForceStrategyOne(strategy).isInternalCall()` instead of reading the storage slot.

Developer Response

Fixed in vfat-tools/sickle-contracts#134.

2. Medium - Fees can be bypassed by using a custom sickle

Strategies use the FeesLib.sol contract to charge fees.

The `chargeFees()` function in this contract can be tricked to not pay any fees if the strategies are used with a custom Sickle.

Technical Details

All strategies have 2 entry points, one that is supposed to be called by the user directly and one that is a callback from the previous call that is executed by the user's sickle as a `delegatecall()`.

But many of the strategies functionalities can be used by directly calling the callback entry point, a user could then deploy its own version of the sickle contract and use the strategies by directly `delegatecall()` to them.

In its own sickle version of the contract, the user could bypass fees using multiple ways:

- by changing the `registry` address to one that would return 0 as fee.
- By changing the `collector` address to itself to just receive the fees directly.

- By changing the `strategy` parameter to something that is not saved in the official `registry` which would result in 0 as a fee.

Here is a POC using the first and second bypass options (it can be copied and pasted in AerodromeStrategy.t.sol:

```
contract BadSickle {
    address public owner;
    address public registry;
    address public collector;

    constructor(address _owner) {
        owner = _owner;
        registry = address(this);
        collector = address(this);
    }

    function deposit(
        address strategy,
        bytes calldata data
    ) public returns (bool success) {
        (success,) = strategy.delegatecall(data);
    }

    function feeRegistry(bytes32 feeHash) public pure returns (uint256) {
        return 0;
    }
}

...

function test_deposit_DepositsUsingDAINoFees()
        public
        prank
        returns (uint256 depositedAmount)
    {
        //deal dai and deploy custom sickle
        uint256 amountIn = 1000 ether;
        BadSickle badSickle = new BadSickle(sickleOwner);

        deal(Base.DAI, sickleOwner, amountIn);
```

```
        IERC20(Base.DAI).approve(address(badSickle), amountIn);

        //prepare deposit call
        IRouter.Route[] memory routeToOther = new IRouter.Route[](1);
        routeToOther[0] = IRouter.Route({
            from: Base.DAI,
            to: Base.WETH,
            stable: false,
            factory: Base.AERODROME_FACTORY
        });

        AerodromeStrategy.ZapInData memory zapData = AerodromeStrategy.ZapInData({
            router: Base.AERODROME_ROUTER,
            tokenIn: Base.DAI,
            amountIn: amountIn,
            routeToIntermediate: new IRouter.Route[](0),
            intermediateMinAmountOut: amountIn,
            routeToOther: routeToOther,
            otherMinAmountOut: 0,
            lpToken: Base.VAMM_WETH_DAI,
            lpTokenMinAmountOut: 0,
            isStablePool: false
        });

        //call deposit
        bytes memory data = abi.encodeCall(
            aerodromeStrategy.AERODROME_DEPOSIT,
            (address(aerodromeStrategy), Base.WETH_DAI_GAUGE, zapData)
        );
        badSickle.deposit(address(aerodromeStrategy), data);

        depositedAmount =
            IGauge(Base.WETH_DAI_GAUGE).balanceOf(address(badSickle));

        //We can see that we were able to deposit and pay no fees to the official
collector
```

```
        assertGt(depositedAmount, 0);
        assertEq(IERC20(Base.DAI).balanceOf(ctx.registry.collector()), 0);
    }
```

Impact

Low/Medium. While most users will use the official Sickle, some could bypass fees while continuing to use the strategies made by the Sickle team.

Recommendation

- Add the `collector` and `registry` as `immutable` variables in the `FeesLib.sol` contract.
- Add a default fee in the `registry` that is returned if the fee doesn't exist (ex: 5%) or consider reverting.

Developer Response

partially fixed in vfat-tools/sickle-contracts#133.

I don't think we really have a need for a default fee so won't be implementing it at this time. It would additionally complicate having operations with a fee set to 0 since we can't easily differentiate a fee that is unset vs explicitly set to 0.

Low Findings

1. Low - No withdraw function in Sickle

Technical Details

While all strategies send back the tokens used to the user/sickle's owner, the Sickle lacks a withdraw function in cases where some tokens are stuck on it.

While it is possible to whitelist some tokens, not all of them will be whitelisted making it hard to withdraw tokens from the contract.

Impact

Low/Informational. Unlikely to have tokens stuck on the contract.

Recommendation

Add a `withdraw()` function limited to the `owner` address.

Developer Response

Acknowledged. This was removed from the core Sickle contracts to simplify them, but the plan is to implement the functionality as a strategy contract if the need arises in the future.

2. Low - If no `routeToOther` is passed to `_zapIn()` `routeToIntermediate` tokens are stuck in the contract

Technical Details

If no `routeToOther` is passed to the `_zapIn()` function, the intermediate tokens received from the `routeToIntermediate` swap will be held as a balance in the Sickle contract.

Impact

Low. This problem only arises when the contract is misused, which is unlikely to occur and can be fixed by zapping from the intermediate route to the other token in another transaction.

Recommendation

Check that if a `routeToIntermediate` is passed to `_zapIn()`, there is a corresponding `routeToOther` also passed, otherwise, revert.

Developer Response

Fixed in vfat-tools/sickle-contracts#139.

3. Low - The `swapInAmount` in `AerodromeStrategy` `_zapIn()` is incorrect for stable pools

Aerodrome uses different k invariants depending on whether the pool is a stable pool or not, which implies that the `swapInAmount` calculated in `_zapIn()` will be incorrect in some cases.

Technical Details

Stable pools in Uniswap, and Uniswap forks, like Aerodrome, have a different curve for stable pools than regular pools as indicated by this comment in Aerodrome's `Pool.sol` (see `_k()` for implementation details). Further, pools can have different fee amounts set in the factory which may also change the calculation needed to determine the expected `swapInAmount` when zapping into a pool.

Impact

Low. Token balances will accrue in the Sickle, but as of now, there does not exist a way for an attacker to get these tokens out of the Sickle.

Recommendation

Re-work the `swapInAmount` calculation in `_zapIn()` in the `AerodromeStrategy` to account for these edge cases.

Developer Response

Fixed by using offchain `swapInAmount` param in e77f907c64cbd3b411cd4ba10fb000bd64f0594b.

4. Low - `msg.sender == sickle.approved()` will always return `false` in `DFORCE_CLAIMDFTOKENREWARDS()`

In the DForceStrategyOne.sol the callback function `DFORCE_CLAIMDFTOKENREWARDS()` has a wrong check that will most likely always return `false`.

Technical Details

The function `DFORCE_CLAIMDFTOKENREWARDS()` is a callback function called by the Sickle after a user called the initial function `dForce_claimDFTokenRewards()`.

At the end of the function, there is a check `msg.sender == sickle.approved()` but because this is a callback function, it is a `delegatecall()` so that means that the `msg.sender` will be the strategy.

So unless the sickle has the strategy address as `approved` it will always return `false`.

Impact

Low. Rewards might be left on the sickle if `isSweepingToOwner` is not set to `true` by the approved address.

Recommendation

Consider setting `isSweepingToOwner` to `true` in the initial function `dForce_claimDFTokenRewards()` if the caller is the approved address.

Developer Response

Fixed in vfat-tools/sickle-contracts#140.

Gas Saving Findings

1. Gas - Make the flashloan strategy variables immutable

Technical Details

In the FlashloanStrategy.sol contract, multiple variables are set only in the constructor and never updated.

Impact

Gas.

Recommendation

Set variables not updated outside the `constructor()` to `immutable`.

Developer Response

Fixed in vfat-tools/sickle-contracts#116.

2. Gas - Useless extra storage of params in the flashloan strategy

Technical Details

In the FlashloanStrategy.sol contract, the `initiateFlashloan()` function save the hash of the `params` in the `flashloanDataHash` storage variable.

But later in this same function the `flashloanDataHash` is updated with a hash of the `params` and `sickleAddress` or with uniswap extra variables making the first save useless.

Impact

Gas. Use extra gas to compute the hash and save it in storage.

Recommendation

Consider removing this initial storage as all flashloan options will overwrite it, and when not going into a flashloan option the call reverts.

Developer Response

Fixed in vfat-tools/sickle-contracts#141.

3. Gas - Useless internal WETH transfer

Technical Details

In the MasterChefStrategy.sol and AerodromeStrategy.sol contracts, the `DEPOSIT` callback functions mint and transfer WETH from the Sickle to itself. Since the WETH contracts send back the tokens to the `msg.sender` this transfer is not needed.

Impact

Gas. Save a WETH transfer.

Recommendation

Remove the `SafeTransferLib.safeTransfer()` after minting WETH.

Developer Response

Fixed in vfat-tools/sickle-contracts#135 and vfat-tools/sickle-contracts#148.

4. Gas - Use router return value to save gas in `DFORCE_SWAPANDLEVERAGEWITHFLASHLOAN()`

Technical Details

In the function `DFORCE_SWAPANDLEVERAGEWITHFLASHLOAN()` of the DForceStrategyOne.sol contract, the variable `capitalAmount` is determined by subtracting `sickleBalanceBeforeSwap` from `sickleBalanceAfterSwap`.

This requires two `balanceOf()` calls to determine how many tokens we received from the swap, but it could be replaced by getting the return value from the router saving two external calls.

Impact

Gas. Save two external calls.

Recommendation

Get the amount received from the router call.

Developer Response

Fixed in vfat-tools/sickle-contracts#142.

5. Gas - Useless `isInternalCall` in DForceStrategyTwo.sol

Technical Details

The contract DForceStrategyTwo.sol has an internal variable `isInternalCall` that is updated by the different functions but never read making it useless.

Impact

Gas/Informational.

Recommendation

Remove the variable from storage and update the functions of the strategy.

Developer Response

Fixed in vfat-tools/sickle-contracts#144.

6. Gas - Change IBalancerVault to save gas

Technical Details

In the FlashloanStrategy.sol the function `initiateFlashloan()` when taking a flashloan from Balance.

It makes a loop into `assets` to convert the `address` to `ERC20`, changing the interface with an array of `address` would use less gas.

Impact

Gas. (ex: save 700~ gas on a 2 assets loop)

Recommendation

Update the interface with an array of `address` and remove the loop.

Developer Response

Fixed in vfat-tools/sickle-contracts#137.

7. Gas - Use unchecked to increment the `i` in `FlashloanStrategy`'s `_setWhitelistedFlashloanOpsSelectors()` for loop

Technical Details

An unchecked block can be used to save gas in `_setWhitelistedFlashloanOpsSelectors()`'s for loop.

Impact

Gas.

Recommendation

Put the `++i;` in an `unchecked` block as is done elsewhere in the code.

```
+       for (uint256 i = 0; i < whitelistedOpsSelectors.length;) {
            if (
                whitelistedFlashloanOpsRegistry[whitelistedOpsSelectors[i]]
                    != address(0)
            ) {
                revert SelectorAlreadyLinked();
            }

            whitelistedFlashloanOpsRegistry[whitelistedOpsSelectors[i]] =
                correspondingStrategies[i];

            emit FlashloanStrategyEvents.SelectorLinked(
                whitelistedOpsSelectors[i], correspondingStrategies[i]
            );
+           unchecked {
+               ++i;
+           }
        }
```

Developer Response

Fixed in vfat-tools/sickle-contracts#143.

Informational Findings

1. Informational - Re: `calculatePremiums()` could revert if AAVEv2 fee is `0`

Technical Details

Similar to the low#4 of the previous report in Aave V2, the implementation can be updated, and the fee changed by the governance. If this is set to 0 the premium calculation for the AAVEv2 pool would revert due to underflow.

Impact

Informational. It's unlikely that the variable will be set to 0.

Recommendation

Check that `aaveV2FlashloanPremiumInBasisPoints > 0` before calculating the premium.

Developer Response

Fixed in vfat-tools/sickle-contracts#145.

2. Informational - Allow user to set a `Referrer` in Masterchef strategy

Technical Details

The MasterChefStrategy.sol contract allows interactions with MasterChef contracts that take a `referrer` address in their params.

But it sets this variable to `address(0)` by default not leveraging the possibility for the user to use a custom address and receive a bonus.

Impact

Informational. Users would benefit from being able to set their own `referrer` variable.

Recommendation

Add a `referrer` variable in the strategy functions params.

Developer Response

Fixed in vfat-tools/sickle-contracts#147.

3. Informational - No Expiration Deadline and `minLiquidityOut` on Aerodrome and Masterchef Strategies

Technical Details

Interactions with AMMs usually allow adding an expiration deadline for the swap to execute but the different functions of the AerodromeStrategy.sol and MasterChefStrategy.sol contracts use `block.timestamp` which result in no deadline.

Additionally, no `minOutA` and `minOutB` is passed in the `addLiquidity` when zapping in, which could lead to sandwich attacks.

Impact

Informational. Auditors couldn't find how validators or MEV bots could profit from these missing parameters.

Recommendation

Consider adding `deadline`, `minOutA`, and `minOutB` parameters to the functions of these two strategies.

Developer Response

Acknowledged. Since this doesn't seem to have any impact, it will not be addressed at this time.

4. Informational - Implementation not initialized on deployment

Technical Details

The Sickle contract has a `initialize()` function that needs to be called to assign an `owner`.

When looking at the `constructor()` and the scripts to deploy the contracts, it seems like the default implementation is not initialized.

Impact

Informational. Auditors couldn't find a way to do any harm by taking ownership of the implementation.

Recommendation

Even if the protocol is not in danger, we strongly advise adding an `initialize()` call to the deployment scripts.

Developer Response

Fixed in vfat-tools/sickle-contracts#138.

Final remarks

The update introduces interesting changes with an overall better code infrastructure. The Sickle team applied the advice given during the previous audit to make their codebase safer.