Mushroom Finance Smart Contract Audit

January 2021

By CoinFabrik

Introduction

CoinFabrik was asked to audit the contracts for the Mushroom project. First we will provide a summary of our discoveries and then we will show the details of our findings.

Summary

The contracts audited are from the Mushroom project. The audit is based on the deployed addresses and the code verified on Etherscan block explorer.

Contracts

The audited contracts are:

File	Address
ControllerV3.sol	0x4bf5059065541a2b176500928e91fbfd0b121d07
MasterChef.sol	0xf8873a6080e8dbf41ada900498de0951074af577
MMToken.sol	0xa283aa7cfbb27ef0cfbcb2493dd9f4330e0fd304
MMVault.sol	0x23b197dc671a55f256199cf7e8bee77ea2bdc16d
StrategyCmpdUsdcV1.sol	0x8f288a56a6c06ffc75994a2d46e84f8bda1a0744
StrategyCurve3CRVv1.sol	0x1f11055eb66f2bba647fb1adc64b0dd4e0018de7

The following contracts are out of the scope:

File	Address
OneSplitAudit.sol	0xc586bef4a0992c495cf22e1aeee4e446cecdee0e
Timelock.sol	0x5dae9b27313670663b34ac8bffd18825bb9df736

For contracts out of scope we assume they will behave according to their interfaces and that malicious parties cannot alter their behavior.

Description

The Mushroom contracts are a fork of the [Yearn]() contracts to which the MasterChef and Flash loan functionalities were added.

We used the contracts descriptions from the Yearn [documentation]().

MasterChef.sol:

The MasterChef contract allows the user to stake MM , LP tokens[1], or MTokens and returns the MM tokens to the user during the time the tokens where staked. The MM tokens are used for governance.

ControllerV3.sol:

The Controller acts as the gatekeeping interface between vaults and strategies and oversees communication and fund flows. Deposits and withdrawals in and out of strategies flow through the Controller. It keeps track of the addresses for the active vaults, strategies, tokens, and strategy rewards destination, acting as a pseudo-registry that verifies the origin and destination of a transaction. The Controller also handles strategy migration, moving funds from an old strategy to a new one.

MMVault.sol:

Vaults act as the representation of the user in the system, and is the internal customer for investments. There is one vault per deposit token, and they are agnostic to the strategies they interact with.

Their primary tasks are to:

- Process user deposits and withdrawals: minting or burning LP tokens as receipts for these;
- Manage disposable funds: Ensuring there is enough to satisfy the minimum amount available to handle withdrawals, and issuing withdrawal requests from strategies when more funds need to be added; and
- Deposit funds into strategies: When there is a surplus of funds in the vault above what's required to be kept at disposal.

[1] Uniswap MM-USDC , Sushiswap MM-WETH, Sushiswap MM-K3PR

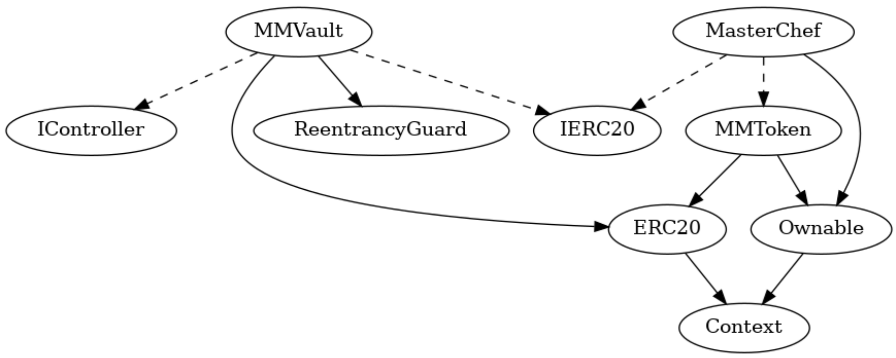
Strategies:

Strategies are investment instruction sets, written by a Strategist. They are agnostic to the vaults that use them. In this audit we will analyze two strategies:

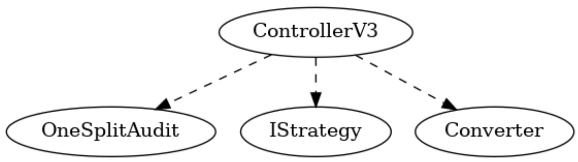
- **CmpdUsdcV1.sol**:
  - USDC tokens are supplied to Compound to earn interest and COMP tokens.
  - USDC tokens are borrowed to be reinvested and earn more COMP tokens.
  - COMP tokens are traded at Uniswap for USDC tokens.
  - USDC tokens are reinvested in Compound.
- **Curve3CRVV1.sol**:
  - 3CRV tokens are supplied to Curve to earn CRV tokens.
  - CRV tokens are traded at Uniswap for DAI, USDC or USDT.
  - That stablecoin is then supplied to the tri-pool in exchange for 3CRV tokens.
  - 3CRV tokens are reinvested in Curve.

Main Contracts

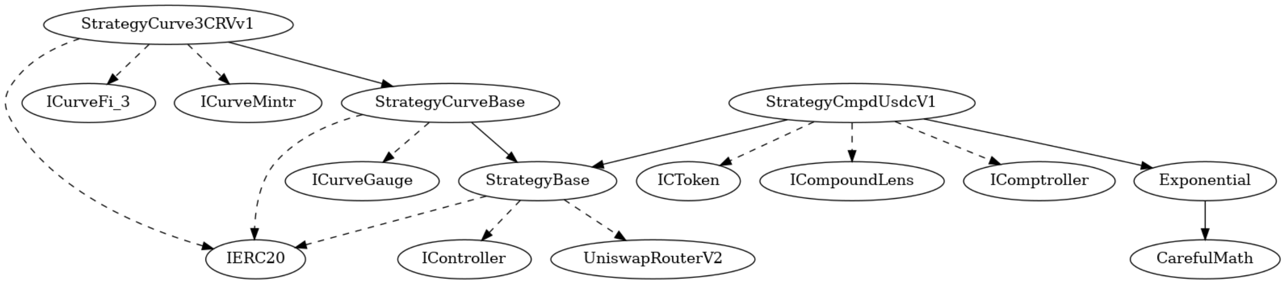
Note: The dashed lines indicate composition and the solid lines indicate inheritance.



Controller



Strategies



Analyses

The following analyses were performed:

- Misuse of the different call methods
- Integer overflow errors
- Division by zero errors
- Outdated version of Solidity compiler
- Front running attacks
- Reentrancy attacks
- Misuse of block timestamps
- Softlock denial of service attacks
- Functions with excessive gas cost
- Missing or misused function qualifiers
- Needlessly complex code and contract interactions
- Poor or nonexistent error handling
- Failure to use a withdrawal pattern
- Insufficient validation of the input parameters
- Incorrect handling of cryptographic signatures

Detailed findings

Severity Classification

Security risks are classified as follows:

- **Critical:** These are issues that we manage to exploit. They compromise the system seriously. They must be fixed **immediately**.
- **Medium:** These are potentially exploitable issues. Even though we did not manage to exploit them or their impact is not clear, they might represent a security risk in the near future. We suggest fixing them **as soon as possible**.
- **Minor:** These issues represent problems that are relatively small or difficult to take advantage of but can be exploited in combination with other issues. These kinds of issues do not block deployments in production environments. They should be taken into account and be fixed **when possible**.
- **Enhancement:** These kinds of findings do not represent a security risk. They are best practices that we suggest to implement.

This classification is summarized in the following table:

SEVERITY	EXPLOITABLE	ROADBLOCK	TO BE FIXED
Critical	Yes	Yes	Immediately
Medium	In the near future	Yes	As soon as possible
Minor	Unlikely	No	Eventually
Enhancement	No	No	Eventually

Issues Found by Severity

Critical severity

No issues of critical severity have been found.

Medium severity

No issues of medium severity have been found.

Minor severity

Ignoring possible errors returned by external calls

In the function getSuppliedView at StrategyCmpdUsdcV1 the possible error code returned by a call to ICToken.getAccountSnapshot is ignored.

```
function getSuppliedView() public view returns (uint256) {
     (, uint256 cTokenBal, , uint256 exchangeRate) = ICToken(cusdc)
            .getAccountSnapshot(address(this));
```

The signature of function getAccountSnapshot at [USDC contract]() indicates that the first parameter returned is an error code, which is ignored by the call from getSuppliedView.

```
/**
 * @notice Get a snapshot of the account's balances, and the cached
exchange rate
 * @dev This is used by comptroller to more efficiently perform
liquidity checks.
 * @param account Address of the account to snapshot
 * @return (possible error, token balance, borrow balance, exchange rate
mantissa)
 */
function getAccountSnapshot(address account) external view returns
(uint, uint, uint, uint) {
```

Similarly, other calls to external functions that may return an invalid value are ignored. For example, in function getMarketColFactor the first parameter from the call to Comptroller is ignored

```
    function getMarketColFactor() public view returns (uint256) {
        (, uint256 colFactor) =
IComptroller(comptroller).markets(cusdc);

        return colFactor;
    }
```

The public getter markets return a struct whose first parameter indicates if the parameter is vali

```
struct Market {
    /// @notice Whether or not this market is listed
    bool isListed;

    /**
     * @notice Multiplier representing the most one can borrow against
their collateral in this market.
     *  For instance, 0.9 to allow borrowing 90% of collateral value.
     *  Must be between 0 and 1, and stored as a mantissa.
     */
    uint collateralFactorMantissa;

    [..]
}

/**
 * @notice Official mapping of cTokens -> Market metadata
 * @dev Used e.g. to determine if a market is supported
 */
mapping(address => Market) public markets;
```

These calls to external functions are invoked from *view* functions that cannot change the contract's internal state. But, it can affect other contracts that invoke those public functions and rely on the values returned.

We suggest to always check the error codes returned by external calls.

Enhancements

Compiler version not fixed

The solidity files use semantic version to indicate they support any solc version above 0.6.7.

It is recommended to explicitly set a fixed version to minimize changes introduced by future untested versions.

```
pragma solidity 0.6.7;
```

Some require statements can be changed to modifiers

The following require statement can be seen in many functions through the code:

```
require(msg.sender == governance, "!governance");
```

The functions using them can be refactored to use a modifier instead

```
modifier onlyGovernance {
      require(msg.sender == governance, "!governance");
      _;
}

[...]
function setDevFund(address _devfund) public onlyGovernance {
      devfund = _devfund;
}
```

The same can be done with the following require statement

```
require(msg.sender == strategist || msg.sender == governance);
```

Like this

```
modifier onlyGovernanceOrStrategist {
      require(msg.sender == strategist || msg.sender == governance);
      _;
```

```
}

[···]
function inCaseTokensGetStuck(address _token, uint _amount) public
onlyGovernanceOrStrategist {
        IERC20(_token).safeTransfer(msg.sender, _amount);
}
```

Observations

Privileged Accounts are EOA

The owner and governance addresses are privileged accounts that can make significant modifications to some of the contracts parameters. In the deployed contracts they are External Owned Accounts, meaning they are controlled by a private key, which creates a possible single point of failure. In the hypothetical case that the private key associated falls in the wrong hands it can cause significant damage to the project.

We suggest using a multisig account to minimize risks until proper governance contracts are developed.

Conclusion

The contracts are simple but interactions with other contracts are complex. They are based on other well known projects like Yearn Finance. The project technical documentation is not abundant.

Our only concern is using EOA for governance and ownership. An easy solution until governance contracts are developed is to use a multisig account and split the responsibility among several members of the Mushroom Finance project.

We found one minor issue that could be easily fixed. We did not discover any critical ones. We suggested a couple of enhancements that will not affect the contract security.

Disclaimer: This audit report is not a security warranty, investment advice, or an approval of the Mushroom Finance project since CoinFabrik has not reviewed its platform. Moreover, it does not provide a smart contract code faultlessness guarantee.