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

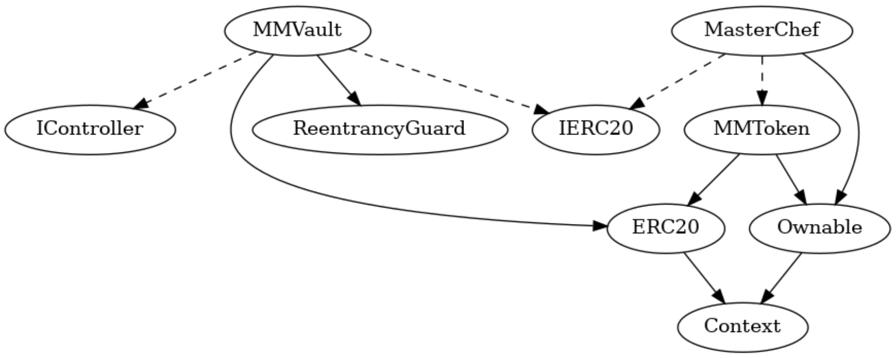
Strategies:

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:

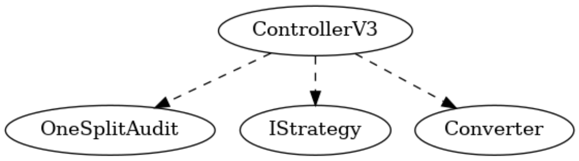
- **CmpdUsdcV1.sol**:
  - USDC tokens are supplied to Compound to earn interest and COMP tokens.
  - USDC tokens are borrowed to be reinvested and earn more COMP tokens.
  - COMP tokens are traded at Uniswap for USDC tokens.
  - USDC tokens are reinvested in Compound.
- **Curve3CRVV1.sol**:
  - 3CRV 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

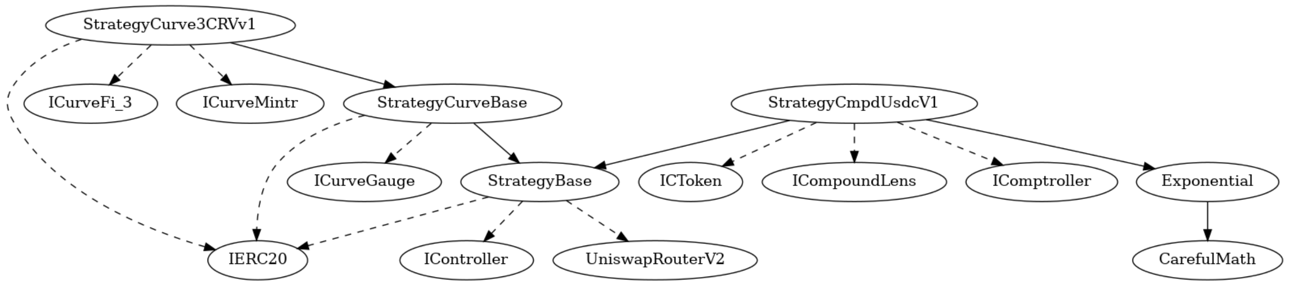
Note: The dashed lines indicate composition and the solid lines indicate inheritance.



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

Denial of Service attack on Compound strategy withdrawals

In the *withdraw* function the balance of USDC is calculated and compared with the amount intended to be withdrawn. In case the balance is insufficient, the withdrawSome function is called with the difference, amount - balance, as parameter.

```
function withdraw(uint256 _amount) external {
    require(msg.sender == controller, "!controller");
    uint256 _balance = IERC20(want).balanceOf(address(this));
    if (_balance < _amount) {
        _amount = _withdrawSome(_amount.sub(_balance));
        _amount = _amount.add(_balance);
    }
```

Then, the _*withdrawSome* function should free up that difference in the borrowed amount for reaching the intended balance asked to be withdrawn.

Instead it asks for USDC balance, compares it with the _*amount* parameter and makes the subtraction again, mistakenly assuming that is the total amount and not the difference.

```
function _withdrawSome(uint256 _amount)internal override returns (uint256) {
    uint256 _want = balanceOfWant();
    if (_want < _amount) {
        uint256 _redeem = _amount.sub(_want);
```

As a consequence the strategy contract will have insufficient funds and the safeTransfer to the vault will revert.

This only happens when the withdrawal amount is greater than the vault USDC balance plus the strategy USDC balance.

In order to make an effective attack, a malicious actor would just need to send some USDC to the **CmpdUsdcV1** contract causing withdrawals (greater than vault's balance) to fail.

However withdrawals will only fail as long as there is some USDC in the contract, the moment those funds get deposited in Compound withdrawals resumes functioning normally.

One more permanent solution we propose is to follow the same semantics used in other strategies in which the _*amount* parameter in _*withdrawSome* is the amount to be redeemed.

```
function _withdrawSome(uint256 _amount)
    internal
    override
    returns (uint256)
{
    // -- CoinFabrik: save initial balance --
    uint256 _balance = balanceOfWant();
    uint256 _redeem = _amount;

    // Make sure market can cover liquidity
    require(ICToken(cusdc).getCash() >= _redeem,
"!cash-liquidity");

    // How much borrowed amount do we need to free?
    uint256 borrowed = getBorrowed();
    uint256 supplied = getSupplied();
    uint256 curLeverage = getCurrentLeverage();
    uint256 borrowedToBeFree = _redeem.mul(curLeverage).div(1e18);

    // If the amount we need to free is > borrowed
    // Just free up all the borrowed amount
    if (borrowedToBeFree > borrowed) {
        this.deleverageToMin();
    } else {
       // Otherwise just keep freeing up borrowed amounts until
       // we hit a safe number to redeem our underlying
       this.deleverageUntil(supplied.sub(borrowedToBeFree));
    }
    // Redeems underlying
    require(ICToken(cusdc).redeemUnderlying(_redeem) == 0,
"!redeem");

    // -- CoinFabrik: calculate tokens redeemed --
    uint256 _reedemed = balanceOfWant();
    _reedemed = _reedemed.sub(_balance);

    return _reedemed;
}
```

This solution was implemented and the issue was fixed in the new version of the smart contract.

StrategyCmpdUsdcV1.sol **at** 0x1a2AAf3bDfce246c6d2F9d93bEe2C649EBE2C32F

Medium severity

No medium severity issues 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 valid

```
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);
}
```

Reduce deployment gas in ControllerV3

In the function *earn* at ControllerV3 there are calls to safeTransfer in both branches of an if statement.

```
function earn(address _token, uint256 _amount) public {
    [..]
    if (_want != _token) {
        [..]
        IERC20(_want).safeTransfer(_strategy, _amount);
    } else {
        IERC20(_token).safeTransfer(_strategy, _amount);
    }
    IStrategy(_strategy).deposit();
}
```

These calls can be merged together since both calls are exactly the same. This change will slightly reduce the cost of deployment without altering the runtime behavior.

```
function earn(address _token, uint256 _amount) public {
    [..]
    if (_want != _token) {
        [..]
    }
    IERC20(_want).safeTransfer(_strategy, _amount);
    IStrategy(_strategy).deposit();
}
```

Use existing function balanceOfWant

Several functions *withdraw*, *withdrawAll*, *harvest* and *deposit* at StrategyCmpdUsdcV1 retrieve the balance calling directly to *want* contract.

```
IERC20(want).balanceOf(address(this))
```

This code can be replaced by a call to *balanceOfWant* that does exactly the same.

```
function balanceOfWant() public view returns (uint256) {
    return IERC20(want).balanceOf(address(this));
}
```

For example the function *deposit* will look like this.

```
function deposit() public override {
    // -- CoinFabrik: use balanceOfWant --
    uint256 _want = balanceOfWant();

    if (_want > 0) {
        IERC20(want).safeApprove(cusdc, 0);
        IERC20(want).safeApprove(cusdc, _want);
        require(ICToken(cusdc).mint(_want) == 0, "!deposit");
    }
}
```

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.

We found a critical issue in the StrategyCmpdUsdcV1 contract that might cause denial of service to the withdrawals. We suggest a solution and urge the team to take measures before it is exploited.

Another 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 critical issue plus a minor one. They do not present a risk of losing funds. The issues could be fixed using the solutions we proposed. We also suggest several enhancements that will not affect the contract security.