SMART CONTRACT CODE REVIEW AND SECURITY ANALYSIS REPORT

Customer: Francovilla
Date: March 19th, 2021

Hacken OÜ
Parda 4, Kesklinn, Tallinn,
10151 Harju Maakond, Eesti,
Kesklinna, Estonia
support@hacken.io

Document

Name	Smart Contract Code Review and Security Analysis Report for Francovilla
Approved by	Andrew Matiukhin | CTO Hacken OU
Type	Token\Staking
Platform	Ethereum / Solidity
Methods	Architecture Review, Functional Testing, Computer-Aided Verification, Manual Review
Deployed contracts	MasterChef: https://bscscan.com/address/0x0bb9a456df06676b8aa133503a9a74c0079fabd3#code HawToken: https://bscscan.com/address/0x709792ad37107cd45f79873d7532216c14a36f33#code
Timeline	18 MARCH 2021 – 19 MARCH 2021
Changelog	19 MARCH 2021 – INITIAL AUDIT 20 MARCH 2021 – SECOND REVIEW

Hacken OÜ
Parda 4, Kesklinn, Tallinn,
10151 Harju Maakond, Eesti,
Kesklinna, Estonia
support@hacken.io

Table of contents

Introduction

Hacken OÜ (Consultant) was contracted by Francovilla (Customer) to conduct a Smart Contract Code Review and Security Analysis. This report presents the findings of the security assessment of Customer's smart contract and its code review conducted on March 18th, 2021.

Scope

The scope of the project is smart contracts deployed in the Binance smart chain network:

```
MasterChef:
https://bscscan.com/address/0x0bb9a456df06676b8aa133503a9a74c0079fabd3#code
HawToken:
https://bscscan.com/address/0x709792ad37107cd45f79873d7532216c14a36f33#code
```

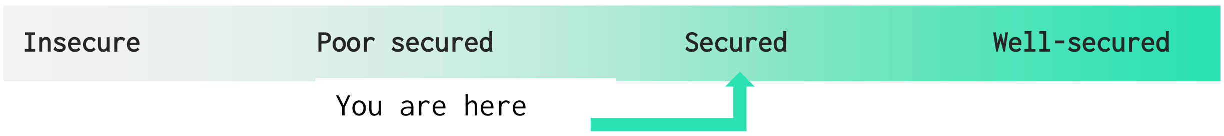
We have scanned this smart contract for commonly known and more specific vulnerabilities. Here are some of the commonly known vulnerabilities that are considered:

Category	Check Item
Code review	▪ Reentrancy ▪ Ownership Takeover ▪ Timestamp Dependence ▪ Gas Limit and Loops ▪ DoS with (Unexpected) Throw ▪ DoS with Block Gas Limit ▪ Transaction-Ordering Dependence ▪ Style guide violation ▪ Costly Loop ▪ ERC20 API violation ▪ Unchecked external call ▪ Unchecked math ▪ Unsafe type inference ▪ Implicit visibility level ▪ Deployment Consistency ▪ Repository Consistency ▪ Data Consistency

Functional review	▪ Business Logics Review ▪ Functionality Checks ▪ Access Control & Authorization ▪ Escrow manipulation ▪ Token Supply manipulation ▪ Asset's integrity ▪ User Balances manipulation ▪ Kill-Switch Mechanism ▪ Operation Trails & Event Generation

Executive Summary

According to the assessment, the Customer's smart is secured.

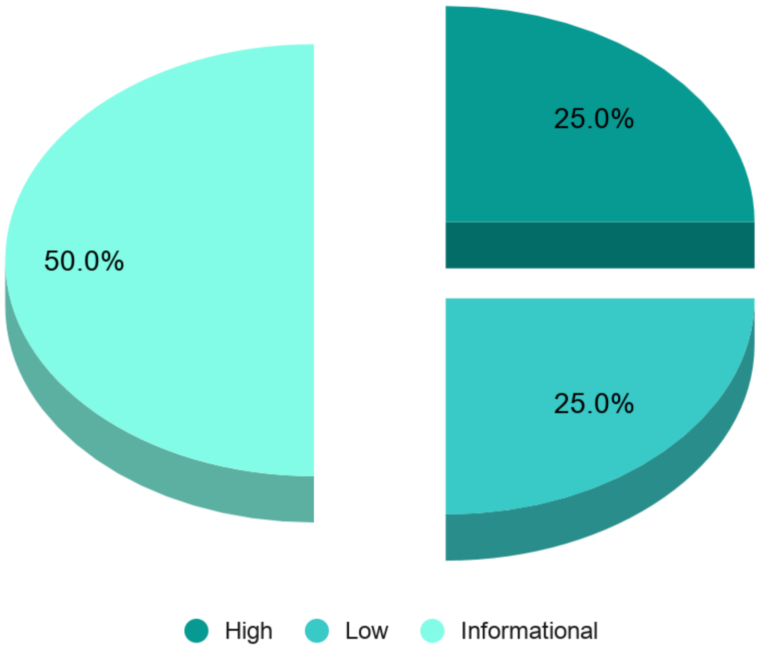


Our team performed an analysis of code functionality, manual audit, and automated checks with Mythril and Slither. All issues found during automated analysis were manually reviewed, and important vulnerabilities are presented in the Audit overview section. A general overview is presented in AS-IS section, and all found issues can be found in the Audit overview section.

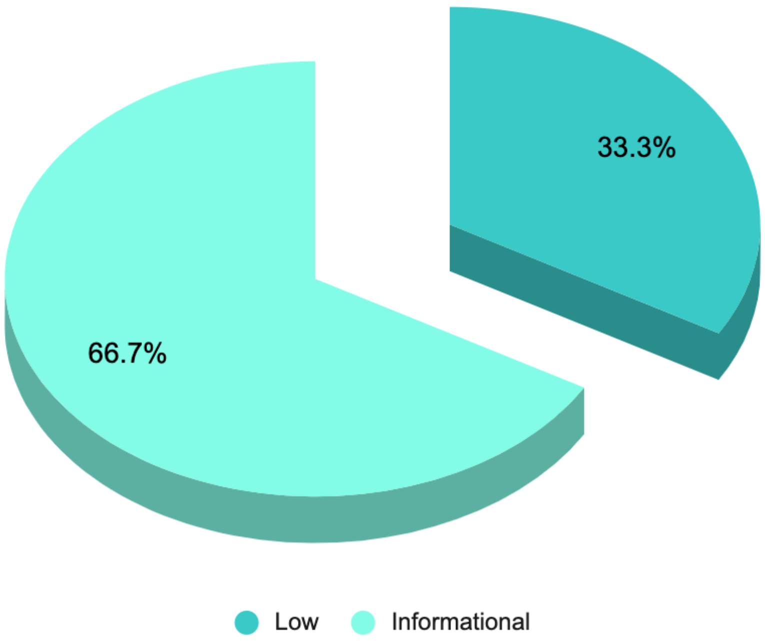
Security engineers found **1** high, **1** low and **2** informational issues during the first review.

After the second review, security engineers found **1** low and **2** informational issues.

Graph 1. The distribution of vulnerabilities after the first review.



Graph 1. The distribution of vulnerabilities after the second review.



Severity Definitions

Risk Level	Description
Critical	Critical vulnerabilities are usually straightforward to exploit and can lead to assets loss or data manipulations.
High	High-level vulnerabilities are difficult to exploit; however, they also have a significant impact on smart contract execution, e.g., public access to crucial functions
Medium	Medium-level vulnerabilities are important to fix; however, they can't lead to assets loss or data manipulations.
Low	Low-level vulnerabilities are mostly related to outdated, unused, etc. code snippets that can't have a significant impact on execution
Lowest / Code Style / Best Practice	Lowest-level vulnerabilities, code style violations, and info statements can't affect smart contract execution and can be ignored.

Audit overview

■ ■ ■ ■ Critical

No Critical severity issues were found.

■ ■ ■ High

1. **Vulnerability:** The *mint* function of the *HawkToken* contract allows the owner to mint an unrestricted amount of tokens to anyone at any time.
   Contract: HawkToken
   Method: mint(address,uint256)

   Fixed before second review by transferring ownership to the MasterChef contract in the tx:
   0x8df2afe4bd017fed1b6c96b060eb6e9dc8606b3b6c525518dcf75f035364dafc

■ ■ Medium

No Medium severity issues were found.

■ Low

1. **Vulnerability:** Reentrancy leading to out-of-order events
   Contract: MasterChef
   Methods:
   - ➢ deposit(uint256,uint256)
   - ➢ emergencyWithdraw(uint256)
   - ➢ enterStaking(uint256)
   - ➢ leaveStaking(uint256)
   - ➢ withdraw(uint256,uint256)

   If the method re-enters, events will be shown in an incorrect order, which might lead to issues for third parties.

   Recommendation: Apply the check-effects-interactions pattern

   Lines: MasterChef.sol#194-201

```
function deposit(uint256 _pid, uint256 _amount) public {
    require (_pid != 0, 'deposit HAWK by staking');
    PoolInfo storage pool = poolInfo[_pid];
    UserInfo storage user = userInfo[_pid][msg.sender];
    updatePool(_pid);
    if (user.amount > 0) {
        uint256 pending =
user.amount.mul(pool.accHawkPerShare).div(1e12).sub(user.rewardDebt);
```

```
        if(pending > 0) {
            safeHawkTransfer(msg.sender, pending);
        }
    }
    if (_amount > 0) {
        pool.lpToken.safeTransferFrom(address(msg.sender),
address(this), _amount);
        user.amount = user.amount.add(_amount);
    }
    user.rewardDebt = user.amount.mul(pool.accHawkPerShare).div(1e12);
    emit Deposit(msg.sender, _pid, _amount);
}
```

Lines: MasterChef.sol#269-276

```
function emergencyWithdraw(uint256 _pid) public {
    PoolInfo storage pool = poolInfo[_pid];
    UserInfo storage user = userInfo[_pid][msg.sender];
    pool.lpToken.safeTransfer(address(msg.sender), user.amount);
    emit EmergencyWithdraw(msg.sender, _pid, user.amount);
    user.amount = 0;
    user.rewardDebt = 0;
}
```

Lines: MasterChef.sol#229-247

```
function enterStaking(uint256 _amount) public {
    require(block.number >= startBlock + 144000); //5 days after Farm
start
    PoolInfo storage pool = poolInfo[0];
    UserInfo storage user = userInfo[0][msg.sender];
    updatePool(0);
    if (user.amount > 0) {
        uint256 pending =
user.amount.mul(pool.accHawkPerShare).div(1e12).sub(user.rewardDebt);
        if(pending > 0) {
            safeHawkTransfer(msg.sender, pending);
        }
    }
    if(_amount > 0) {
        pool.lpToken.safeTransferFrom(address(msg.sender),
address(this), _amount);
        user.amount = user.amount.add(_amount);
    }
    user.rewardDebt = user.amount.mul(pool.accHawkPerShare).div(1e12);

    emit Deposit(msg.sender, 0, _amount);
}
```

Lines: MasterChef.sol#250-266

```
function leaveStaking(uint256 _amount) public {
    PoolInfo storage pool = poolInfo[0];
    UserInfo storage user = userInfo[0][msg.sender];
    require(user.amount >= _amount, "withdraw: not good");
    updatePool(0);
    uint256 pending =
user.amount.mul(pool.accHawkPerShare).div(1e12).sub(user.rewardDebt);
    if(pending > 0) {
        safeHawkTransfer(msg.sender, pending);
    }
    if(_amount > 0) {
        user.amount = user.amount.sub(_amount);
        pool.lpToken.safeTransfer(address(msg.sender), _amount);
    }
    user.rewardDebt = user.amount.mul(pool.accHawkPerShare).div(1e12);

    emit Withdraw(msg.sender, 0, _amount);
}
```

Lines: MasterChef.sol#209-226

```
function withdraw(uint256 _pid, uint256 _amount) public {
    require (_pid != 0, 'withdraw HAWK by unstaking');
    PoolInfo storage pool = poolInfo[_pid];
    UserInfo storage user = userInfo[_pid][msg.sender];
    require(user.amount >= _amount, "withdraw: not good");

    updatePool(_pid);
    uint256 pending =
user.amount.mul(pool.accHawkPerShare).div(1e12).sub(user.rewardDebt);
    if(pending > 0) {
        safeHawkTransfer(msg.sender, pending);
    }
    if(_amount > 0) {
        user.amount = user.amount.sub(_amount);
        pool.lpToken.safeTransfer(address(msg.sender), _amount);
    }
    user.rewardDebt = user.amount.mul(pool.accHawkPerShare).div(1e12);
    emit Withdraw(msg.sender, _pid, _amount);
}
```

Lowest / Code style / Best Practice

1. **Vulnerability:** Conformance to Solidity naming conventions
   Contract: MasterChef

   Solidity defines a naming convention that should be followed.

   Lines: MasterChef.sol#95

```
function add(uint256 _allocPoint, IBEP20 _lpToken, bool _withUpdate)
public onlyOwner {
```

   Lines: MasterChef.sol#111

```
function set(uint256 _pid, uint256 _allocPoint, bool _withUpdate)
public onlyOwner {
```

   Lines: MasterChef.sol#287

```
function updateHawkPerBlock(uint256 _hawkPerBlock) external onlyOwner {
```

2. **Vulnerability:** Public function that could be declared external
   Contract: MasterChef

   public functions that are never called by the contract should be declared **external** to save gas.

   Lines: MasterChef.sol#85

```
function updateMultiplier(uint256 multiplierNumber) public onlyOwner {
```

   Lines: MasterChef.sol#95

```
function add(uint256 _allocPoint, IBEP20 _lpToken, bool _withUpdate)
public onlyOwner {
```

   Lines: MasterChef.sol#111

```
function set(uint256 _pid, uint256 _allocPoint, bool _withUpdate)
public onlyOwner {
```

   Lines: MasterChef.sol#184

```
function deposit(uint256 _pid, uint256 _amount) public {
```

   Lines: MasterChef.sol#203

```
function deposited(uint256 _pid) public view returns(uint256) {
```

Lines: MasterChef.sol#250

```
function leaveStaking(uint256 _amount) public {
```

Lines: MasterChef.sol#269

```
function emergencyWithdraw(uint256 _pid) public {
```

Lines: MasterChef.sol#294

```
function dev(address _devaddr) public {
```

Conclusion

Smart contracts within the scope were manually reviewed and analyzed with static analysis tools. For the contract, high-level description of functionality was presented in As-Is overview section of the report.

Audit report contains all found security vulnerabilities and other issues in the reviewed code.

Security engineers found **1** high, **1** low and **2** informational issues during the audit.

After a second review security engineers found **1** low and **2** informational issues.

Category	Check Items	Comments
Code Review	Style guide violation	Public function that could be declared external and naming convention
	Data Consistency	Re-entrancy which can lead to out-of-order events

Disclaimers

Hacken Disclaimer

The smart contracts given for audit have been analyzed in accordance with the best industry practices at the date of this report, in relation to cybersecurity vulnerabilities and issues in smart contract source code, the details of which are disclosed in this report (Source Code); the Source Code compilation, deployment, and functionality (performing the intended functions).

The audit makes no statements or warranties on security of the code. It also cannot be considered as a sufficient assessment regarding the utility and safety of the code, bugfree status or any other statements of the contract. While we have done our best in conducting the analysis and producing this report, it is important to note that you should not rely on this report only - we recommend proceeding with several independent audits and a public bug bounty program to ensure security of smart contracts.

Technical Disclaimer

Smart contracts are deployed and executed on blockchain platform. The platform, its programming language, and other software related to the smart contract can have its vulnerabilities that can lead to hacks. Thus, the audit can't guarantee the explicit security of the audited smart contracts.