





Unicrypt Farm Contracts
SMART CONTRACT AUDIT

05.11.2020

Made in Germany by Chainsulting.de



Table of contents

1. Disclaimer.....	3
2. About the Project and Company	4
2.1 Project Overview.....	5
3. Vulnerability & Risk Level	6
4. Auditing Strategy and Techniques Applied.....	7
4.1 Methodology	7
4.2 Used Code from other Frameworks/Smart Contracts	8
4.3 Tested Contract Files	9
4.4 Metrics / CallGraph.....	10
4.5 Metrics / Source Lines	11
5. Scope of Work & Results.....	12
5.1 Manual and Automated Vulnerability Test.....	13
5.1.1 Wrong import of OpenZeppelin library	13
5.1.2 Improved Logic in getMultiplier	14
5.1.3 Missing natspec documentation.....	15
5.2. SWC Attacks	16
5.3. Special Checks	20
5.3.1 Test deployment	20
5.3.2 Function: Withdraw 	22
5.3.3 Function: Emergency Withdraw 	23
6. Executive Summary.....	24
7. Deployed Smart Contract	25



1. Disclaimer

The audit makes no statements or warranties about utility of the code, safety of the code, suitability of the business model, investment advice, endorsement of the platform or its products, regulatory regime for the business model, or any other statements about fitness of the contracts to purpose, or their bug free status. The audit documentation is for discussion purposes only.

The information presented in this report is confidential and privileged. If you are reading this report, you agree to keep it confidential, not to copy, disclose or disseminate without the agreement of Unicrypt.network. If you are not the intended receptor of this document, remember that any disclosure, copying or dissemination of it is forbidden.

Major Versions / Date	Description
0.1 (01.11.2020)	Layout and details (Metrics / Scope of work)
0.5 (01.11.2020)	Automated Security Testing Manual Security Testing
0.8 (02.11.2020)	Adding of SWC, Special Checks
1.0 (03.11.2020)	Final document (Summary and Recommendation)
1.5 (05.11.2020)	Status change of bugs
1.6 (05.11.2020)	Adding Mainnet addresses

2. About the Project and Company

Company address: NA (ANON)

Website: <https://unicrypt.network/>

GitHub: NA

Twitter: https://twitter.com/UNCX_token

Telegram: https://t.me/uncx_token

Etherscan (UNCX Token): <https://etherscan.io/token/0xaDB2437e6F65682B85F814fBc12FeC0508A7B1D0>

Medium: <https://unicrypt.medium.com/>

2.1 Project Overview

The Unicrypt platform allows yield farming virtually any ERC20 token. It provides safe vault contracts for other tokens to deposit the farm rewards into, and a dApp that's targeted for mobile and desktop use with connections to all major wallets for users to farm their favourite tokens on.

3. Vulnerability & Risk Level

Risk represents the probability that a certain source-threat will exploit vulnerability, and the impact of that event on the organization or system. Risk Level is computed based on CVSS version 3.0.

Level	Value	Vulnerability	Risk (Required Action)
Critical	9 – 10	A vulnerability that can disrupt the contract functioning in a number of scenarios, or creates a risk that the contract may be broken.	Immediate action to reduce risk level.
High	7 – 8.9	A vulnerability that affects the desired outcome when using a contract, or provides the opportunity to use a contract in an unintended way.	Implementation of corrective actions as soon as possible.
Medium	4 – 6.9	A vulnerability that could affect the desired outcome of executing the contract in a specific scenario.	Implementation of corrective actions in a certain period.
Low	2 – 3.9	A vulnerability that does not have a significant impact on possible scenarios for the use of the contract and is probably subjective.	Implementation of certain corrective actions or accepting the risk.
Informational	0 – 1.9	A vulnerability that have informational character but is not effecting any of the code.	An observation that does not determine a level of risk

4. Auditing Strategy and Techniques Applied

Throughout the review process, care was taken to evaluate the repository for security-related issues, code quality, and adherence to specification and best practices. To do so, reviewed line-by-line by our team of expert pentesters and smart contract developers, documenting any issues as there were discovered.

4.1 Methodology

The auditing process follows a routine series of steps:

1. Code review that includes the following:
 - i. Review of the specifications, sources, and instructions provided to Chainsulting to make sure we understand the size, scope, and functionality of the smart contract.
 - ii. Manual review of code, which is the process of reading source code line-by-line in an attempt to identify potential vulnerabilities.
 - iii. Comparison to specification, which is the process of checking whether the code does what the specifications, sources, and instructions provided to Chainsulting describe.
2. Testing and automated analysis that includes the following:
 - i. Test coverage analysis, which is the process of determining whether the test cases are actually covering the code and how much code is exercised when we run those test cases.
 - ii. Symbolic execution, which is analysing a program to determine what inputs causes each part of a program to execute.
3. Best practices review, which is a review of the smart contracts to improve efficiency, effectiveness, clarify, maintainability, security, and control based on the established industry and academic practices, recommendations, and research.
4. Specific, itemized, actionable recommendations to help you take steps to secure your smart contracts.

4.2 Used Code from other Frameworks/Smart Contracts

1. SafeMath.sol (0.6.0)

<https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/math/SafeMath.sol>

2. IERC20.sol (0.6.0)

<https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/token/ERC20/IERC20.sol>

3. Ownable.sol (0.6.0)

<https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/access/Ownable.sol>

4. ERC20.sol (0.6.0)

<https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/token/ERC20/ERC20.sol>

5. SafeERC20.sol (0.6.0)

<https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/token/ERC20/SafeERC20.sol>

6. Context.sol (0.6.0)

<https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/GSN/Context.sol>

7. Address.sol (0.6.0)

<https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/utils/Address.sol>

8. EnumerableSet.sol (0.6.0)

<https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/utils/EnumerableSet.sol>

9. TransferHelper (0.6.0)

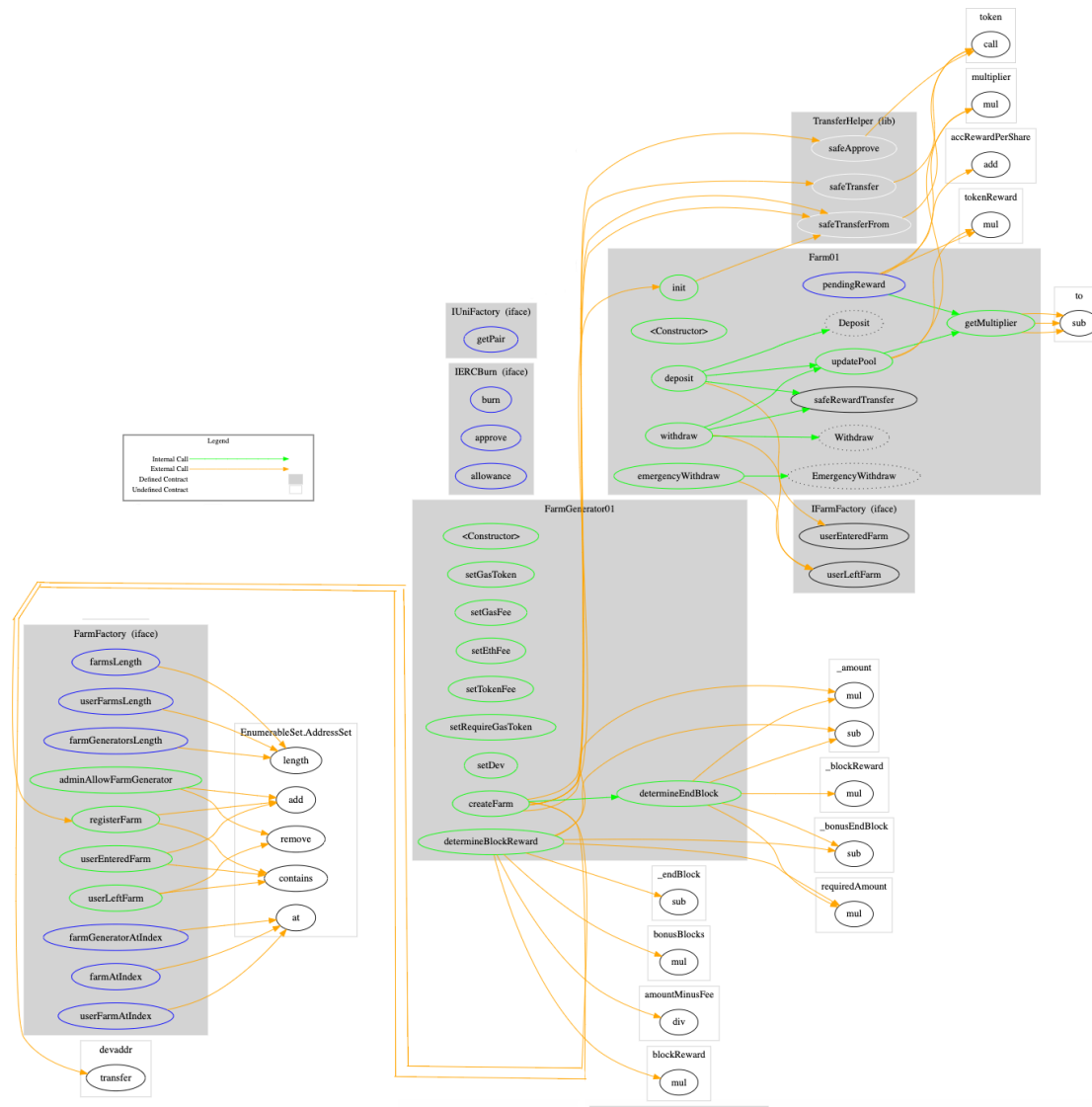
<https://github.com/Uniswap/uniswap-lib/blob/master/contracts/libraries/TransferHelper.sol>

4.3 Tested Contract Files

The following are the SHA-256 hashes of the reviewed files. A file with a different SHA-256 hash has been modified, intentionally or otherwise, after the security review. You are cautioned that a different SHA-256 hash could be (but is not necessarily) an indication of a changed condition or potential vulnerability that was not within the scope of the review

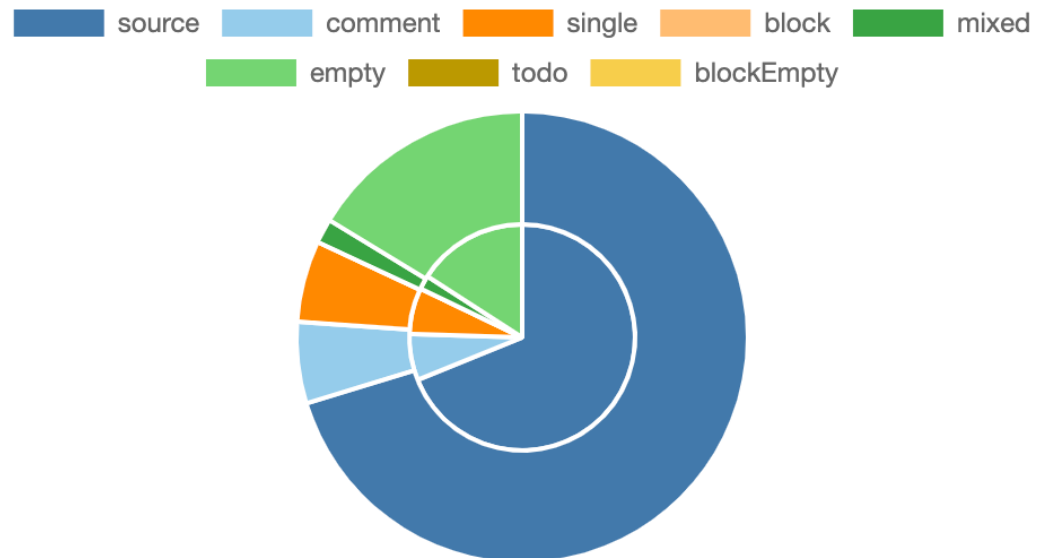
File	Fingerprint (SHA256)
Farm01.sol	336835b41a7251629d0906cf8bd469063167f8b5d7914eb05c9a503ec8670f0a
FarmFactory.sol	a80474494cc756af769e80d3292ad874ff0fb3aec0ed0f7138348cdb835c749e
FarmGenerator01.sol	bd99347194811489fb59bc68482e3f06add83d9da5beba8dfa558696533ad69a

4.4 Metrics / CallGraph



4.5 Metrics / Source Lines

Source Lines (sloc vs. nsloc)



5. Scope of Work & Results

The UniCrypt team provided us with the files that needs to be tested. The scope of the audit is FarmFactory.sol, FarmGenerator01.sol and Farm01.sol contracts with its direct imports.

The team put forward the following assumptions regarding the security of the FarmFactory.sol, FarmGenerator01.sol and Farm01.sol Audit contract:

- Yield farmer are always able to withdraw their tokens.
- The function 'emergencyWithdraw' works.
- The function 'emergencyWithdraw' ('farmInfo.numFarmers--;') fail if farmInfo.numFarmers is 0.

The main goal of this audit was to verify these claims and check the overall security of the codebase.

5.1 Manual and Automated Vulnerability Test

CRITICAL ISSUES

During the audit, Chainsulting's experts found **no Critical issues** in the code of the smart contract.

HIGH ISSUES

During the audit, Chainsulting's experts found **no High issues** in the code of the smart contract.

MEDIUM ISSUES

5.1.1 Wrong import of OpenZeppelin library

Severity: Medium

Status: **Fixed**

File(s) affected: FarmGenerator01.sol, FarmFactory.sol, Farm01.sol

Attack / Description	Code Snippet	Result/Recommendation
In the current implementation, OpenZeppelin files are added to the repo. This violates OpenZeppelin's MIT license, which requires the license and copyright notice to be included if its code is used. Moreover, updating code manually is error-prone.	NA	We highly recommend using npm (import "@openzeppelin/contracts/..") in order to guarantee that original OpenZeppelin contracts are used with no modifications. This also allows for any bug-fixes to be easily integrated into the codebase.

LOW ISSUES

5.1.2 Improved Logic in getMultiplier

Severity: LOW

Status: **Fixed**

File(s) affected: Farm01.sol

Line: 78 - 89

Attack / Description	Code Snippet	Result/Recommendation
<p>The function getMultiplier takes two arguments, i.e., _from and _to, and calculates the reward multiplier for the given block range ([_from, _to]). We notice that this helper does not take into account the initial block (startBlock) from which the incentive rewards start to apply. As a result, when a normal user gives arbitrary arguments, it could return wrong reward multiplier! A correct implementation needs to take startBlock into account and appropriately re-adjusts the starting block number, i.e., _from = _from >= startBlock ? _from : startBlock.</p>	<pre>// Return reward multiplier over the given _from to _to block. function getMultiplier(uint256 _from, uint256 _to) public view returns (uint256) { uint256 to = farmInfo.endBlock > _to ? _to : farmInfo.endBlock; if (to <= farmInfo.bonusEndBlock) { return to.sub(_from).mul(farmInfo.bonus); } else if (_from >= farmInfo.bonusEndBlock) { return to.sub(_from); } else { return farmInfo.bonusEndBlock.sub(_from).mul(farmInfo. bonus).add(to.sub(farmInfo.bonusEndBlock)); } }</pre>	<p>Apply additional sanity checks in the getMultiplier() routine so that the internal _from parameter can be adjusted to take startBlock into account.</p>

INFORMATIONAL ISSUES

5.1.3 Missing natspec documentation

Severity: INFORMATIONAL

Status: **Fixed**

File(s) affected: FarmGenerator01.sol, FarmFactory.sol, Farm01.sol

Attack / Description	Code Snippet	Result/Recommendation
Solidity contracts can use a special form of comments to provide rich documentation for functions, return variables and more. This special form is named the Ethereum Natural Language Specification Format (NatSpec).	NA	It is recommended to include natspec documentation and follow the doxygen style including @author, @title, @notice, @dev, @param, @return and make it easier to review and understand your smart contract.

5.2. SWC Attacks

ID	Title	Relationships	Test Result
SWC-131	Presence of unused variables	CWE-1164: Irrelevant Code	✓
SWC-130	Right-To-Left-Override control character (U+202E)	CWE-451: User Interface (UI) Misrepresentation of Critical Information	✓
SWC-129	Typographical Error	CWE-480: Use of Incorrect Operator	✓
SWC-128	DoS With Block Gas Limit	CWE-400: Uncontrolled Resource Consumption	✓
SWC-127	Arbitrary Jump with Function Type Variable	CWE-695: Use of Low-Level Functionality	✓
SWC-125	Incorrect Inheritance Order	CWE-696: Incorrect Behavior Order	✓
SWC-124	Write to Arbitrary Storage Location	CWE-123: Write-what-where Condition	✓
SWC-123	Requirement Violation	CWE-573: Improper Following of Specification by Caller	✓

ID	Title	Relationships	Test Result
SWC-122	Lack of Proper Signature Verification	CWE-345: Insufficient Verification of Data Authenticity	✓
SWC-121	Missing Protection against Signature Replay Attacks	CWE-347: Improper Verification of Cryptographic Signature	✓
SWC-120	Weak Sources of Randomness from Chain Attributes	CWE-330: Use of Insufficiently Random Values	✓
SWC-119	Shadowing State Variables	CWE-710: Improper Adherence to Coding Standards	✓
SWC-118	Incorrect Constructor Name	CWE-665: Improper Initialization	✓
SWC-117	Signature Malleability	CWE-347: Improper Verification of Cryptographic Signature	✓
SWC-116	Timestamp Dependence	CWE-829: Inclusion of Functionality from Untrusted Control Sphere	✓
SWC-115	Authorization through tx.origin	CWE-477: Use of Obsolete Function	✓
SWC-114	Transaction Order Dependence	CWE-362: Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition')	✓


ID	Title	Relationships	Test Result
SWC-113	DoS with Failed Call	CWE-703: Improper Check or Handling of Exceptional Conditions	✓
SWC-112	Delegatecall to Untrusted Callee	CWE-829: Inclusion of Functionality from Untrusted Control Sphere	✓
SWC-111	Use of Deprecated Solidity Functions	CWE-477: Use of Obsolete Function	✓
SWC-110	Assert Violation	CWE-670: Always-Incorrect Control Flow Implementation	✓
SWC-109	Uninitialized Storage Pointer	CWE-824: Access of Uninitialized Pointer	✓
SWC-108	State Variable Default Visibility	CWE-710: Improper Adherence to Coding Standards	✓
SWC-107	Reentrancy	CWE-841: Improper Enforcement of Behavioral Workflow	✓
SWC-106	Unprotected SELFDESTRUCT Instruction	CWE-284: Improper Access Control	✓
SWC-105	Unprotected Ether Withdrawal	CWE-284: Improper Access Control	✓
SWC-104	Unchecked Call Return Value	CWE-252: Unchecked Return Value	✓


ID	Title	Relationships	Test Result
SWC-103	Floating Pragma	CWE-664: Improper Control of a Resource Through its Lifetime	✓
SWC-102	Outdated Compiler Version	CWE-937: Using Components with Known Vulnerabilities	✓
SWC-101	Integer Overflow and Underflow	CWE-682: Incorrect Calculation	✓
SWC-100	Function Default Visibility	CWE-710: Improper Adherence to Coding Standards	✓


5.3. Special Checks

5.3.1 Test deployment

1. Deployment of FarmFactory.sol 

2. Deployment of FarmGenerator01.sol with arguments (address _FACTORY, address _FARMGENERATOR: ANY).
To bypass the Uniswap contract check we commented the block 174-177 and make the testing easier, therefore the second argument in the constructor for FarmGenerator01.sol can be any address. 

3. Now we need to call 'adminAllowFarmGenerator' on the FarmFactory.sol contract with arguments (address FarmGenerator01, true) 

4. Creating a farm via interaction with FarmGenerator01.sol. The 'createFarm' function will deploy farm vaults by locking the amount of tokens you send to it and only allowing it to be farmed out over the given period with the specified Uniswap LP tokens. 

createFarm ^

_rewardToken:

address

_amount:

uint256

_lpToken:

address

_blockReward:

uint256

_startBlock:

uint256

_bonusEndBlock:

uint256

_bonus:

uint256

5.3.2 Function: Withdraw

Resources:

1. https://fravoll.github.io/solidity-patterns/checks_effects_interactions.html#:~:text=In%20an%20unsafe%20implementation%20of,contract%20could%20reenter%20our%20function
2. <https://docs.openzeppelin.com/contracts/2.x/api/token/erc20#SafeERC20>

Code:

```
function withdraw(uint256 _amount) public {
    UserInfo storage user = userInfo[msg.sender];
    require(user.amount >= _amount, "INSUFFICIENT");
    updatePool();
    if (user.amount == _amount && _amount > 0) {
        factory.userLeftFarm(msg.sender);
        farmInfo.numFarmers--;
    }
    uint256 pending = user.amount.mul(farmInfo.accRewardPerShare).div(1e12).sub(user.rewardDebt);
    safeRewardTransfer(msg.sender, pending);
    user.amount = user.amount.sub(_amount);
    user.rewardDebt = user.amount.mul(farmInfo.accRewardPerShare).div(1e12);
    farmInfo.lpToken.safeTransfer(address(msg.sender), _amount);
    emit Withdraw(msg.sender, _amount);
}
```

Result:

The function withdraw is following the checks-effects-interactions pattern. Although the LP tokens are assumed trusted as of now, cause the UniswapV2's LP tokens are not vulnerable or exploitable for re-entrancy.

5.3.3 Function: Emergency Withdraw

Resources:

1. https://fravoll.github.io/solidity-patterns/checks_effects_interactions.html#:~:text=In%20an%20unsafe%20implementation%20of,contract%20could%20reenter%20our%20function
2. <https://docs.openzeppelin.com/contracts/2.x/api/token/erc20#SafeERC20>

Code:

```
function emergencyWithdraw() public {
    UserInfo storage user = userInfo[msg.sender];
    farmInfo.lpToken.safeTransfer(address(msg.sender), user.amount);
    emit EmergencyWithdraw(msg.sender, user.amount);
    if (user.amount > 0) {
        factory.userLeftFarm(msg.sender);
        farmInfo.numFarmers--;
    }
    user.amount = 0;
    user.rewardDebt = 0;
}
```

Result:

The function emergencyWithdraw is following the checks-effects-interactions pattern. Although the LP tokens are assumed trusted as of now, cause the UniswapV2's LP tokens are not vulnerable or exploitable for re-entrancy.

6. Executive Summary

The smart contracts are written as simple as possible and also not overloaded with unnecessary functions. Most functions are widely used by common yield farming contracts and audited several times, these is greatly benefiting the security of the contracts. It correctly implemented widely-used and reviewed contracts from OpenZeppelin and for safe mathematical operations. The main goal of the audit was to verify the claims regarding the security of the smart contract (see the Scope of work section). According to the code, the implementation of this functions considers all security checks for a safe withdrawal of funds.

Both claims appear valid. During the audit, no critical or high issues were found after the manual and automated security testing.

Edit: The Unicrypt Team reacted promptly on our findings and fixed all bugs.

7. Deployed Smart Contract

Deployed Unicrypt Contracts (Mainnet)

FarmFactory.sol

<https://etherscan.io/address/0x388f7E6d45e058AA703227B44e216e3bE3C6A6E7#code> (approved)

FarmGenerator01.sol & Farm01.sol

<https://etherscan.io/address/0x197D2286f299C323272C08D768D7fD987e1350F2#code> (approved)