RAID
SQUARE

Antfarm.Finance

SMART CONTRACT AUDIT REPORT

26 JANUARY 2023

Audited by: RAID Square

RAID Square Audit Report: #2023-001

Table of Contents

1. INTRODUCTION

As part of a request to improve the security of their protocol, Antfarm Finance commissioned RAID Square to carry out an audit of the various smart contracts listed below in the "Mission" section. This document highlights the results of the audit that was carried out and highlights the risks and opportunities for improvement from the current state of the contracts at the time of delivery.

1.1 About RAID Square

RAID Square is a French company specialised in risk mitigation for Blockchain stakeholders. RAID Square's offerings are structured around the following four business verticals:

- Legal structuring and coordination of WEB3 legal procedures ;
- Development & cyber cross WEB2 X WEB3 audits ;
- Due diligence & investigations ;
- Lobbying of French and European institutions ;

1.1 About Antfarm Finance

Antfarm is a decentralised exchange (DEX) designed for cash providers (LPs). It is supported by a decentralised autonomous organisation (DAO). The main objective of Antfarm is to attract, remunerate and retain LPs, increasing their profits while reducing their risks. Antfarm was created to solve the problem that providing liquidity in most DEX pools is not financially attractive for LPs, who often receive little compensation and face the risk of impermanent loss.

The Antfarm protocol provides for the use of the Antfarm Token (ATF) in each pool to pay the exchange fee and allows LPs to access their collected fees at any time without impacting the total locked-in value of the pool. A portion of the claimed fees is burned, resulting in a steady increase in ATF scarcity. Antfarm is governed by a decentralised autonomous organisation (DAO) in which holders of the Antfarm Governance Token (AGT) can participate in decision-making processes and benefit from the growth of the platform.

1. 3 Mission

Item	Description
Project name	antfarm.finance
Website	https://antfarm.finance/
Type	Smart Contract
Detail	Solidity
Mission start date	Wed. 8 Dec. 2022
Mission end date	Mon. 10 Jan. 2023
Version	1.1.0

The RAID Square mission is to carry out audits of the contracts listed below:

- Smart Contract #1: **AntfarmFactory.sol**
  - *https://github.com/AntfarmFinance/contracts/blob/main/antfarm/AntfarmFactory.sol*
- Smart Contract #2: **AntfarmOracle.sol**
  - *https://github.com/AntfarmFinance/contracts/blob/main/antfarm/AntfarmOracle.sol*
- Smart Contract #3: **AntfarmPair.sol**
  - *https://github.com/AntfarmFinance/contracts/blob/main/antfarm/AntfarmPair.sol*
- Smart Contract #4: **AntfarmAtfPair.sol**
  - *https://github.com/AntfarmFinance/contracts/blob/main/antfarm/AntfarmAtfPair.sol*
- Smart Contract #5: **AntfarmPosition.sol**
  - *https://github.com/AntfarmFinance/contracts/blob/main/antfarm/AntfarmPosition.sol*

1. 3 Mission

Smart Contract	Hash sha256
AntfarmAtfPair.sol	**Hash 1.0:** *f874d656b3836f6d708a13917608ac8913fa30e2bf9fedc58c8510afb99df8d7* **Hash 1.1:** 02bfa1702b26d0477262b2b0f92d8049343fbecf03c7cdfb1c9bb365f459aea6
AntfarmPair.sol	**Hash 1.0:** *581f91ff53394b0a337e4549e141d0dfff85b8d3efdb20107627e38b5f9ddd1e* **Hash 1.1:** c74fbce0e15b504189707200d6a4df45768276ad63162f7e3e7afba483bb1548
AntfarmOracle.sol	**Hash 1.0:** *e0a2b7acee9b11e18bfe5843c40877698d7bb03add78da3b9cc7128036e987c8* **Hash 1.1:** 105397d5302f32527755ee7f38a9522513c24f7cea9dfd4b50b05759d91e338c
AntfarmFactory.sol	**Hash 1.0:** *c5bf921eb91d4f35782de00c66308a3c80cd9787fbd7793787c2111cea3e167c* **Hash 1.1:** 6c7902bc14eb1aa5c1e6dd49cb1b09ddad8ebafe730dccface9a082d6058e5fd
AntfarmPosition.sol	**Hash 1.0:** *d2da444b9cd0c7e5cf29f97bc83707ec242e5aa1b27eac744dad37d55ae8b4a0* **Hash 1.1:** 73b7f05f3c6faf96528bf80fefab5a90907b8d857ad851808e4fd4d00babc908

The calculation of the SHA-256 hash of each smart contract as part of this audit may allow verification of the version of the code that was audited by the RAID Square team.

1. 4 Methodology

A smart contract audit is a process of examining and assessing the security and quality of a smart contract, with the aim of identifying vulnerabilities and ensuring that the contract is working as intended. There are several different methodologies that can be used to conduct a smart contract audit, each with its own set of best practices and considerations.

It is important to note that a smart contract audit is not a one-off process, but rather an ongoing effort to ensure that the contract remains secure and functions as intended. Regular audits should be conducted to identify and resolve any issues that may arise over time.

RAID Square has created its audit methodology based on different existing models and bases its approach on the notion of risk.

Risk = Likelihood * Impact

Risk Severity				
Impact	High	Medium	High	Critical
	Medium	Intermediary	Medium	High
	Low	Low	Intermediary	Medium
		High	Medium	Low
	Likelihood			

1. 4 Methodology

The probability estimate and the impact estimate are put together to calculate an overall severity for each risk. This is done by determining whether the probability is low, intermediary, medium, high or c and then doing the same for the impact. The 0 to 10 scale:

Likelihood and Impact Levels	
0 to <2	Low
2 to <4	Intermediary
4 to <6	Medium
6 to <9	High
10	Critical

In order to determine the probability, we use the following guiding factors for each vulnerability found:

Skill Level - The level of skill required to carry out the attack:
- ❏ No technical skills (1 to 3)
- ❏ Some technical skills (3 to 5)
- ❏ Advanced computer user (5 to 6)
- ❏ Networking and programming skills (6 to 9)
- ❏ Security penetration skills (9 to 10)

1. 4 Methodology

Motivation - How motivated is the attacker to find and exploit the vulnerability?

- ❏ No reward (1 to 2)
- ❏ Reward less than the cost of the attack (2 to 4)
- ❏ Reward at break-even cost of attack (4 to 6)
- ❏ Reward greater than the cost of the attack (6 to 9)
- ❏ High or full reward (10)

Opportunity - What resources are required for the attacker to find and exploit this vulnerability?

- ❏ Full access or expensive resources required (0 to 2)
- ❏ Access or special resources required (2 to 7)
- ❏ Some access or few resources required (7 to 9)
- ❏ No access or resources required (9 to 10)

Status - What is the status of the attacker?

- ❏ Developers (2)
- ❏ System administrators (2 to 4)
- ❏ Intranet users (4 to 5)
- ❏ Partners (5 to 6)
- ❏ Authenticated users (6 to 9)
- ❏ Anonymous Users (9 to 10)

Discovery difficulty - How easy is it for the attacker to discover this vulnerability?

- ❏ Nearly impossible (1 to 2)
- ❏ Difficult (3 to 7)
- ❏ Easy (8 to 9)
- ❏ Automated tools available (10)

1. 4 Methodology

Exploitation difficulty - How easy is it for the attacker to actually exploit this vulnerability?

- ❏ Theoretical (1 to 2)
- ❏ Difficult (3 to 4)
- ❏ Easy (5 to 9)
- ❏ Automated tools available (10)

Awareness - To what extent is this vulnerability known to the attacker?

- ❏ Unknown (1 to 3)
- ❏ Hidden (4 to 5)
- ❏ Obvious (6 to 9)
- ❏ Public knowledge (10)

Detection - How likely is this exploit to be detected?

- ❏ Active detection in the application (1 to 2)
- ❏ Logged and reviewed (3 to 7)
- ❏ Logged without review (8 to 9)
- ❏ Not logged (10)

In order to determine the impact, we use the following guiding factors for each vulnerability found:

Confidentiality - How much data could be disclosed and how sensitive is it (if linked to a central database)?

- ❏ Minimum non-sensitive data disclosed (2 to 5)
- ❏ Minimum critical data disclosed (6 to 7)
- ❏ Extended non-sensitive data disclosed (6 to 7)
- ❏ Extended critical data disclosed (7 to 9)
- ❏ All data disclosed (10)

1. 4 Methodology

Integrity - How much data could be corrupted and how badly is it damaged?

- ❏ Minimal data slightly corrupted (1 to 2)
- ❏ Minimal data severely corrupted (3 to 4)
- ❏ Extensive data slightly corrupted (5 to 6)
- ❏ Severely corrupted extended data (7 to 9)
- ❏ All data totally corrupted (10)

Non-compliance - What degree of exposure does non-compliance cause?

- ❏ Minor violation (2 to 4)
- ❏ Open violation (5 to 6)
- ❏ High-profile violation (7 to 10)

Availability - How much service could be lost and how vital is it?

- ❏ Minimal secondary services interrupted (1 to 4)
- ❏ Minimal primary services interrupted (5 to 6)
- ❏ Extended secondary services interrupted (5 to 6)
- ❏ Extended primary services interrupted (7 to 9)
- ❏ All services totally lost (10)

Finance - What will be the financial damage from an exploit?

- ❏ Less than the cost of fixing the vulnerability (1 to 2)
- ❏ Minor effect on annual profit (3 to 6)
- ❏ Major effect on annual profit (7 to 9)
- ❏ Bankruptcy (10)

1. 4 Disclaimer

RAID Square is performing a technical audit of the smart contract code as an independent service provider. We have performed this audit in a diligent and professional manner, but it is important to understand that our job is only to verify the quality and security of the code in its current state.

This security audit is not intended to replace the functional testing required before any software release. Furthermore, it does not guarantee the discovery of all possible security issues of the given smart contract(s) or blockchain software. In other words, the result of the evaluation does not guarantee the non-existence of any other security issues.

As an audit-based evaluation cannot be considered complete, we always recommend several independent audits and a public bug bounty programme to ensure the security of the smart contract(s).

We are not responsible for any security flaws that may be discovered later, nor for any damage that may result from them. Similarly, we are not responsible for any changes made to the code after the audit has been completed.

It is the responsibility of users and stakeholders to exercise due diligence when using the audited smart contracts and to regularly check their security. RAID Square accepts no liability for any loss or damage arising from the use of or reliance on these smart contracts.

Last but not least, this security audit should not be used as investment advice.

2. FINDINGS

2.1 Summary

Here is a summary of our findings after analysing the implementation of the Antfarm Finance protocol. During the first phase of our audit, we studied the source code of the smart contracts using our internal static code analyser. After identifying known coding bugs and checking them manually, we manually examined the logic of the code for potential bugs.

Severity	Number of findings
Low	16
Intermediary	0
Medium	1
High	1
Critical	0
Total	18

3. DETAILED RESULTS

3.1 AntfarmFactory.sol

3.1.1 Floating Compiler Version — Low —> Fixed

Line: `2`

```
pragma solidity ^0.8.10;
```

Description: Contracts must be deployed with the same compiler version and flags as those with which they have been thoroughly tested. Locking the pragma ensures that contracts are not accidentally deployed using, for example, an obsolete compiler version that could introduce bugs that adversely affect the contract system.

Recommendation: Use fixed solidity compiler version.
Code: `pragma solidity =0.8.10;`

3.1.2 Outdated code — Low—> Confirmed

Line: `11, 21, 22, 26, 30, 44, 52, 77`

```
override
```

Description: The version currently used in the smart contract is 0.8.10. Since version 0.8.8, an earlier version, the inheritance of interface functions does not require the “override” specifier. By removing them, it is possible to reduce the size of the contract code.

Recommendation: Remove `override` when it’s to override an interface.

3.1.3 DoS With Block Gas Limit — Medium—> Fixed

Line: 58-60

```
function getAllPairs() external view returns
(address[] memory) {
    return allPairs;
}
```

Description: When smart contracts are deployed or functions within them are called, the execution of these actions always requires a certain amount of gas, depending on the amount of computation needed to perform them. The Ethereum network specifies a gas limit per block and the sum of all transactions included in a block cannot exceed this threshold. Programming schemes that are harmless in centralized applications can lead to denial of service conditions in smart contracts when the cost of executing a function exceeds the block gas limit. Modifying an array of unknown size, which increases in size over time, can lead to such a denial-of-service condition.

Recommendation: All users can increase the size of `allPairs` with the `createPair` function. The `getAllPairs()` function might not work anymore if `allPairs` becomes too large because the reading might consume more gas than the gas block limit. In your case, the `getAllPairs()` function is not used in the contracts we audited, which is why this problem is in the medium state.
It is advised not to return a dynamic array, you can use `allPairsLength()` and call with a specific array id on `allPairs`. If you want to retrieve several pieces of data in one call, you must return a fixed amount of data.

3.2 AntfarmOracle.sol

3.2.1 Floating Compiler Version — Low—> Fixed

Line: 2

```
pragma solidity ^0.8.10;
```

Description: Contracts must be deployed with the same compiler version and flags as those with which they have been thoroughly tested. Locking the pragma ensures that contracts are not accidentally deployed using, for example, an obsolete compiler version that could introduce bugs that adversely affect the contract system.

Recommendation: Use fixed solidity compiler version.

Code:

```
pragma solidity =0.8.10;
```

3.3 AntfarmPair.sol

3.3.1 Floating Compiler Version — Low—> Fixed

Line: `2`

```
pragma solidity ^0.8.10;
```

Description: Contracts must be deployed with the same compiler version and flags as those with which they have been thoroughly tested. Locking the pragma ensures that contracts are not accidentally deployed using, for example, an obsolete compiler version that could introduce bugs that adversely affect the contract system.

Recommendation: Use fixed solidity compiler version.
Code: `pragma solidity =0.8.10;`

3.3.2 Outdated code — Low—> Confirmed

Line: `21, 24, 27, 30, 33, 36, 39, 42, 84, 98, 137, 179, 241, 257, 273, 281, 291, 306, 322, 342, 359`

```
override
```

Description: The version currently used in the smart contract is 0.8.10. Since version 0.8.8, an earlier version, the inheritance of interface functions does not require the “override” specifier. By removing them, it is possible to reduce the size of the contract code.

Recommendation: Remove `override` when it’s to override an interface.

3.3.3 Code readability — Low—> Fixed

Line: `42`

```
AntfarmOracle public override antfarmOracleToken;
```

Description: It is important to keep some readability in the smart contract code, to interact with an external smart contract, we mainly use an interface and an address declaration. For example, on `line 24`: `address public override token0;`, the IERC20 interface is used as a type converter (`IERC20(token0).balanceOf(address(this));`).
For `antfarmOracleToken`, it is directly an import of contract code on `IAntfarmPair` -> `IAntfarmBase` -> `import "../AntfarmOracle.sol";`

Recommendation: Use interface like other external smart contract interaction.

3.3.4 Operator efficiency — Low—> Confirmed

3.3.4.1 Operator efficiency — Low—> Confirmed

Line: `22`

```
totalSupply = totalSupply + liquidity;
```

Description: Code logic error. Logical operators can do the same action but in a clearer way.

Recommendation: Use the `+=` operator.
Code: `totalSupply += liquidity;`

3.3.4.2 Operator efficiency — Low—> Confirmed

Line: 164

```
totalSupply = totalSupply - liquidity;
```

Description: Code logic error. Logical operators can do the same action but in a clearer way.

Recommendation: Use the `-=` operator.
Code: `totalSupply -= liquidity;`

3.3.4.3 Operator efficiency — Low—> Confirmed

Line: 423

```
totalDividendPoints = totalDividendPoints + ((amount
* POINT_MULTIPLIER) / (totalSupply -
MINIMUM_LIQUIDITY));
```

Description: Code logic error. Logical operators can do the same action but in a clearer way.

Recommendation: Use the `+=` operator.
Code:

```
totalDividendPoints += ((amount * POINT_MULTIPLIER)
/ (totalSupply - MINIMUM_LIQUIDITY));
```

3.3.4.5 Operator efficiency — Low—> Confirmed

Line: `424`

```
antfarmTokenReserve = antfarmTokenReserve + amount;
```

Description: Code logic error. Logical operators can do the same action but in a clearer way.

Recommendation: Use the `+=` operator.
Code: `antfarmTokenReserve += amount;`

3.3.5 Code readability — Low—> Confirmed

Line: `367`

```
for (uint256 token; token < 2; ++token) {
```

Description: If a variable is supposed to be initialized to zero, it must be explicitly noted as zero to improve the readability of the code. The `uint256 token` must be set to 0 to verify the 2 tokens of `address[2] memory tokens`
.

Recommendation: Initialize all the variables.
Code:

```
for (uint256 token = 0; token < 2; ++token) {
```

3.3.6 Code readability — Low—> Confirmed

Line: `293`

```
uint112 maxReserve;
```

Description: If a variable is supposed to be initialized to zero, it must be explicitly set to zero to improve the readability of the code. The `uint112 maxReserve;` must be set to 0 for `address bestOracle = scanOracles(maxReserve);`.

Recommendation: Initialize all the variables.
Code: `uint112 maxReserve = 0;`

3.3.7 Oracle Manipulation — High—> Confirmed

Line:

```
179 - 183: function swap(uint256 amount0Out,uint256
amount1Out,address to) external override nonReentrant
{
...
218: uint256 feeToPay;
...
220: feeToPay = getFees(amount0Out, amount0In,
amount1Out, amount1In);
221: if (feeToPay < MINIMUM_LIQUIDITY) revert
SwapAmountTooLow();
```

```
222 - 226:  if
(IERC20(antfarmToken).balanceOf(address(this)) -
antfarmTokenReserve < feeToPay) {
227:  revert InsufficientFee();
228: }
...
233: uint256 feeToDisburse = (feeToPay * 8500) /
10000;
234: uint256 feeToBurn = feeToPay - feeToDisburse;
...
236: _disburse(feeToDisburse);
237: IAntfarmToken(antfarmToken).burn(feeToBurn);
```

Description: If someone manages to manipulate the price through high liquidity in the pair, `feeToPay` might not be legitimate. Attackers could manipulate the price to avoid paying a `feeToPay`. The swap system could be locked if `feeToPay` is too low because `feeToPay` must be higher than `MINIMUM_LIQUIDITY` (`if (feeToPay < MINIMUM_LIQUIDITY) revert SwapAmountTooLow();`). An attacker can use a flash loan, or create a pair and change the oracle to this one

Recommendation: This problem has no direct solution but the risk can be limited by reducing the update time in the oracle (`uint256 public constant PERIOD = 1 hours;`). If someone has enough money to manipulate the price and pay the fee on `AntfarmAtfPair`, the `feeToPay` will not correlate with the actual `antfarmToken` for a short period of time.

3.4 AntfarmAtfPair.sol

3.4.1 Floating Compiler Version — Low—> Fixed

Line: `2`

```
pragma solidity ^0.8.10;
```

Description: Contracts must be deployed with the same compiler version and flags as those with which they have been thoroughly tested. Locking the pragma ensures that contracts are not accidentally deployed using, for example, an obsolete compiler version that could introduce bugs that adversely affect the contract system.

Recommendation: Use fixed solidity compiler version.
Code: `pragma solidity =0.8.10;`

3.4.2 Outdated code — Low—> Confirmed

Line: `20, 23, 26, 29, 32, 35, 38, 41, 88, 98, 143, 187, 248, 262, 280, 293, 303, 319`

```
override
```

Description: The version currently used in the smart contract is 0.8.10. Since version 0.8.8, an earlier version, the inheritance of interface functions does not require the “override” specifier. By removing them, it is possible to reduce the size of the contract code.

Recommendation: Remove `override` when it’s to override an interface.

3.4.3 Code readability— Low—> Confirmed

Line: `41`

```
AntfarmOracle public override antfarmOracle;
```

Description: It is important to keep some readability in the smart contract code, to interact with an external smart contract, we mainly use an interface and an address declaration. For example, on `line 23`: `address public override token0;`, the IERC20 interface is used as a type converter (`IERC20(token0).balanceOf(address(this));`).
For `antfarmOracle`, it is directly an import of contract code on `IAntfarmPair` -> IAntfarmBase -> `import "../AntfarmOracle.sol";`

Recommendation: Use interface like other external smart contract interaction and remove direct smart contract importing interface file.

3.4.4 Operator efficiency — Low—> Confirmed

3.4.4.1 Operator efficiency — Low—> Confirmed

Line: `124`

```
totalSupply = totalSupply + liquidity;
```

Description: Code logic error. Logical operators can do the same action but in a clearer way.

Recommendation: Use the `+=` operator.
Code: `totalSupply += liquidity;`

3.4.4.2 Operator efficiency — Low—> Confirmed

Line: 166

```
totalSupply = totalSupply - liquidity;
```

Description: Code logic error. Logical operators can do the same action but in a clearer way.

Recommendation: Use the `-=` operator.
Code: `totalSupply -= liquidity;`

3.4.4.3 Operator efficiency — Low—> Confirmed

Line: 382

```
totalDividendPoints = totalDividendPoints + ((amount
* POINT_MULTIPLIER) / (totalSupply -
MINIMUM_LIQUIDITY));
```

Description: Code logic error. Logical operators can do the same action but in a clearer way.

Recommendation: Use the `+=` operator.
Code:

```
totalDividendPoints += ((amount * POINT_MULTIPLIER)
/ (totalSupply - MINIMUM_LIQUIDITY));
```

3.4.4.4 Operator efficiency — Low—> Confirmed

Line: 166

```
antfarmTokenReserve = antfarmTokenReserve + amount;
```

Description: Code logic error. Logical operators can do the same action but in a clearer way.

Recommendation: Use the `+=` operator.
Code: `antfarmTokenReserve += amount;`

3.5 AntfarmPosition.sol

3.5.1 Floating Compiler Version — Low—> Fixed

Line: `2`

```
pragma solidity ^0.8.10;
```

Description: Contracts must be deployed with the same compiler version and flags as those with which they have been thoroughly tested. Locking the pragma ensures that contracts are not accidentally deployed using, for example, an obsolete compiler version that could introduce bugs that adversely affect the contract system.

Recommendation: Use fixed solidity compiler version.
Code: `pragma solidity =0.8.10;`

3.5.2 Outdated code — Low—> Fixed

Line: `General`

```
override
```

Description: The version currently used in the smart contract is 0.8.10. Since version **0.8.8**, an earlier version, the inheritance of interface functions does not require the “override” specifier. By removing them, it is possible to reduce the size of the contract code.

Recommendation: Remove `override` when it’s to override an interface.

3.5.3 Code readability — Low—> Confirmed

3.5.3.1 Code readability — Low—> Confirmed

Line: 401

```
for (uint256 i; i < positionsLength; ++i) {
```

Description: If a variable is meant to be initialized to zero, explicitly set it to zero to improve code readability. The `uint256 i;` must be set to 0 to `uint256 i = 0;`

Recommendation: Initialize all variables.
Code:

```
for (uint256 i = 0; i < positionsLength; ++i) {
```

3.5.3.2 Code readability — Low—> Confirmed

Line: 421

```
for (uint256 i; i < positionIds.length; ++i) {
```

Description: If a variable is meant to be initialized to zero, explicitly set it to zero to improve code readability. The `uint256 i;` must be set to 0 to `uint256 i = 0;`

Recommendation: Initialize all variables.
Code:

```
for (uint256 i = 0; i < positionIds.length; ++i) {
```

3.5.3.3 Code readability — Low—> Confirmed

Line: 473

```
for (uint256 i; i < positionsLength; ++i) {
```

Description: If a variable is meant to be initialized to zero, explicitly set it to zero to improve code readability. The `uint256 i;` must be set to 0 to `uint256 i = 0;`

Recommendation: Initialize all variables.
Code:

```
for (uint256 i = 0; i < positionsLength; ++i) {
```