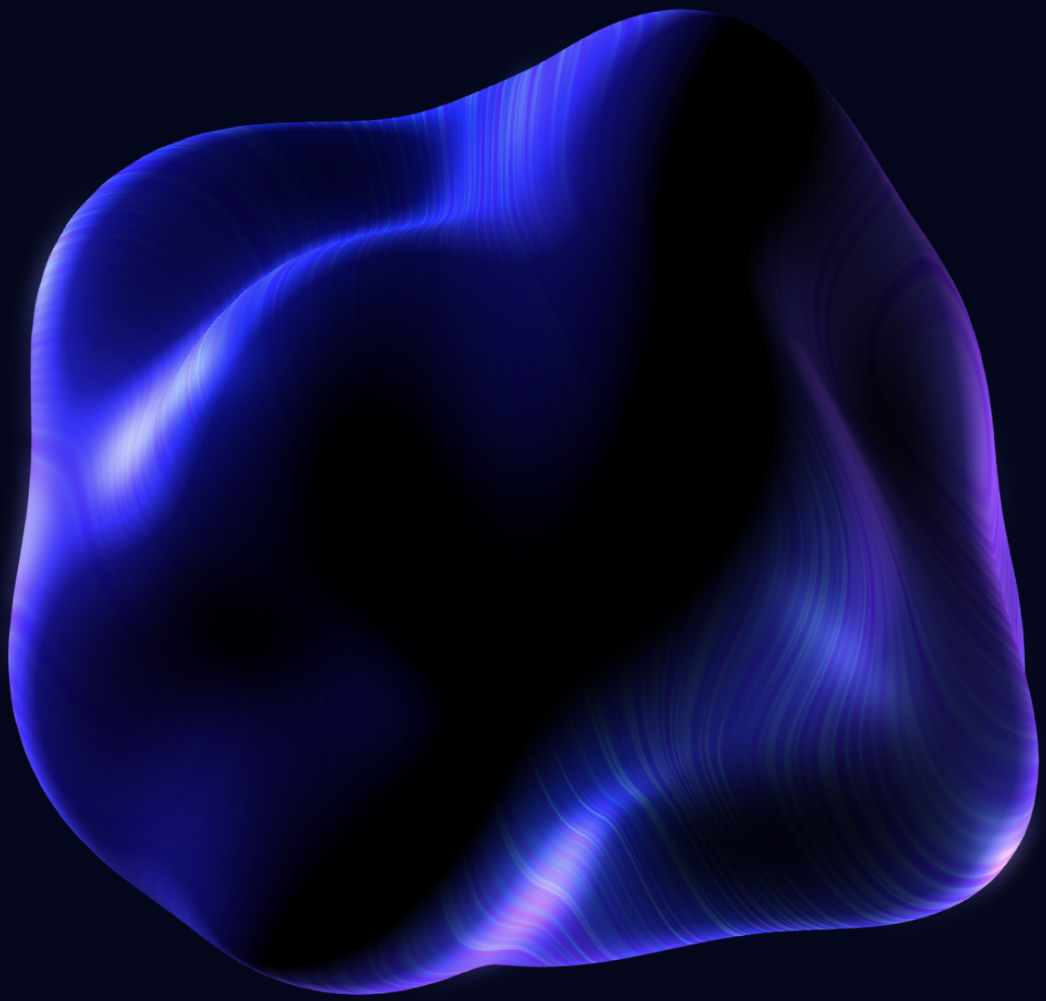
MUNDUS
SECURITY

Security
Smart Contract Audit



MUNDUS
SECURITY

8.Finance Token Report

This document may contain confidential information about IT systems and the intellectual property of the Customer as well as information about potential vulnerabilities and methods of their exploitation.
The report containing confidential information can be used internally by the Customer, or it can be disclosed publicly after all vulnerabilities are fixed — upon a decision of the Customer.

Audit scope

Name	8.Finance
Type	GameFi
Language	Solidity
Repository	https://github.com/8-finance/token-contract
Commit	eea4181cd0ac4bd7e1b6884eb1ae7113383b388d
Technical documentation	https://docs.8.finance/
JS Tests	Unit tests only
Website	https://8.finance/
Timeline	NO
Changelog	NO

MUNDUS
SECURITY

Table of contents

1. Project overview
2. Finding Severity breakdown
3. Summary of findings
4. Findings overview
5. Disclaimers

MUNDUS
SECURITY

Project overview

8F token is a utility and governance ERC20 token of the 8.Finance protocol with in-built deflation logic. Each transfer of 8F burns fraction of tokens transferred which ensures deflation. No minting of 8F tokens is allowed after deployment of the contract.

Finding Severity breakdown

All vulnerabilities discovered during the audit are classified based on their potential severity and have the following classification:

Severity	Description
High	Bugs that can trigger a contract failure or theft of assets. Further recovery is possible only by manual modification of the contract state or replacement.
Medium	Bugs that can break the intended contract logic or expose it to DoS attacks, but do not cause direct loss of funds.
Low	Bugs that do not pose significant danger to the project or its users but are recommended to fixed nonetheless.
Informational	All other non-essential recommendations.
Gas	Gas optimization recommendations.

MUNDUS
SECURITY

Summary of findings

Severity	Found	Acknowledged	Resolved
High	0	0	0
Medium	0	0	0
Low	1	1	0
Informational	2	2	0
Gas	3	3	0
Total	6	6	0

Bugs description

Id	Description	Severity
01	Possible 0 fee `transfer` with lowered `burnTransferFee`	Low
02	Pragma allows suboptimal versions of Solidity	Informational
03	Undefined `burnTransferFee` in deployed token's storage	Informational
04	Assembly division with denominator `>0`	Gas
05	Custom errors instead of revert strings	Gas
06	Unchecked subtraction with impossible underflow	Gas

Findings

[01 - Low] Possible 0 fee `transfer` with lowered `burnTransferFee`

Description

The `_transfer` method of the `Token8FUpgradeableV1` contract invokes `_burn` with each transfer of tokens, if `fee > 0`. In case `amount <= FEE_MULTIPLIER / burnTransferFee - 1`, the fee for the transfer will be zero due to integer arithmetics. If the transfer fee is equal to 1%, the maximum amount of tokens available for transfer without the fee is negligible (`99*10^(-18)` 8F tokens for 1% fee). However, if the fee is lowered, this amount grows as `1 / fee`.

Recommendation

Consider introducing minimal transfer fee state variable and corresponding logic to the `Token8FUpgradeableV1` contract.

[02 - Informational] Pragma allows suboptimal versions of Solidity

Description

Current pragma for the `Token8FUpgradeableV1` contract allows compilation with older versions of Solidity language. This can lead to suboptimal performance of the contract.

Recommendation

Consider changing pragma to `^0.8.4`.

[03 - Informational] Undefined burnTransferFee in deployed token's storage

Description

The burnTransferFee state variable of the current Token8FUpgradeableV1 deployment is unset.

Recommendation

Set the burnTransferFee variable before launching the protocol or consider initializing it in the constructor.

[04 - Gas] Assembly division with denominator >0

Description

The _transfer method of the Token8FUpgradeableV1 contract calculates the transfer fee as uint256 fee = (amount * burnTransferFee) / FEE_MULTIPLIER. Each time division is invoked solidity checks for denominator != 0. In the case of fee calculation this check is superfluous, as the FEE_MULTIPLIER is strictly greater than zero.

Recommendation

Consider using assemble for the fee calculation as proposed in the following codeblock. This will save 100 gas with each transfer.

```
uint256 fee = amount * burnTransferFee;
assembly {
  fee := div(fee, FEE_MULTIPLIER)
}
```

[05 - Gas] Custom errors instead of revert strings

Description

The `setBurnTransferFee` method of the `Token8FUpgradeableV1` contract reverts with an error string if `feePercent >= 100000000`. Since `0.8.4` Solidity allows the use of custom error strings, which are more gas efficient during the deployment of the contract and each time the revert is invoked than the error strings.

Recommendation

Consider introducing custom errors to the contract.

[06 - Gas] Unchecked subtraction with impossible underflow

Description

The amount of tokens transferred to an address after the `_burn` method is invoked is equal to `amount - fee`. As `fee` is always strictly less than the `amount`, underflow check is superfluous.

Recommendation

Consider using unchecked arithmetics for subtraction.

```
unchecked {
  super._transfer(owner, to, amount - fee);
}
```

Disclaimers

Mundus 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 the security of the code. It also cannot be considered as a sufficient assessment regarding the utility and safety of the code, bug-free 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 the security of smart contracts.

Technical disclaimers

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