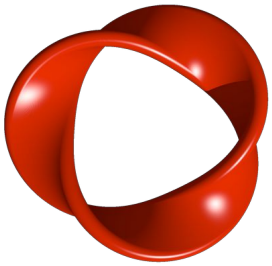

Secfault Security

Review Smart Contracts

Security Assessment

Report

for

Taurus SA

Dr. Jean-Philippe Aumasson
Pl. Ruth-Bösiger 6
1201 Genève

- hereafter called "Taurus" -

Document History

Version	Author	Date	Comment
0.1	Gregor Kopf	2025-02-28	First Draft
0.2	Dirk Breiden	2025-02-28	Internal Review
1.0	Gregor Kopf	2025-03-07	Final Version

Table of Contents

1 Executive Summary

Taurus requested consultation by Secfault Security to perform a security review of two Ethereum smart contract solutions.

The analyzed smart contracts implement an ERC-20 and ERC-721 interface. Both contracts come in a standalone and in an upgradeable version. The solutions are based on the OpenZeppelin[1] smart contract library.

Secfault Security has performed a review of the contracts in a time-frame from 2025-02-24 to 2025-02-28 within an overall effort of two person days, including documentation. This report describes the results of the ERC-20 review.

During the analysis, one observation have been made, which Secfault Security would like to bring to Taurus ' attention. It is not a directly exploitable vulnerability - however, given that fact that the analyzed smart contracts in some sense inherently manage financial assets, Secfault Security believes that a high level of awareness for potential corner-cases is beneficial.

Overall, the codebase left a positive impression. The smart contracts mostly use standard, library-provided functionality, which is generally considered good practice.

1 https://www.openzeppelin.com/

2 Overview

The following sections provide an overview of the project execution, the scope of the assessment as well as a brief summary of the test procedures applied during the engagement.

2.1 Target Scope

The contracts subject to review have been provided by Taurus via their GitHub repository. The following commits have been analyzed:

- TERC-20: `0f1a7782a883c98be0f13375db610d02e956c269`
- TERC-721: `bb5275194cb4aed6068c1d8efc1958b76309a2f7`

2.2 Test Procedures

Secfault Security performed a manual source code review of the provided contracts. While Secfault Security also leverages tool-assisted approaches (such as using slither[2]), a manual inspection of the codebase constitutes the main part of such a project. This is due to the fact that Secfault Security strives to obtain an actual in-depth understanding of the underlying application logic, in order to pinpoint possible logic issues.

The following section provides an exemplary overview of some of the most prominent security issues that occur in Ethereum smart contracts. As these types of flaws can have major impact on the security of the analyzed smart contract, a substantial amount of time was put into evaluating the contracts for these kind of vulnerabilities.

2.2.1 Integer Issues

Older solidity versions suffered, as other major languages like C or C++, from integer-related issues like overflows and underflows. The reviewed codebase however relies on `solidity >=0.8.0`, which is not prone to this problem anymore, except when explicitly using the `unchecked` keyword.

2.2.2 Frontrunning Problems

Frontrunning is an inherent problem of the Ethereum blockchain. It can occur if contracts depend on transactions being processed in order. As a matter of fact, the order of transactions inside a block can be subject to manipulation. Transactions are visible to everyone on the blockchain even before they are included in a block. Given that miners, by default, often order transactions by their gas price and include the most gas-heavy ones first in a block, an attacker can easily observe transactions and place their own transaction in front of specific transactions by setting a higher gas price.

2 https://github.com/crytic/slither

One issue related to frontrunning has been identified during the review. It is described in section 4.1 of this document.

2.2.3 Reentrancy

Calling external contracts comes with the infamous possibility of reentrancy attacks, meaning that a malicious contract calls back into the calling contract in a recursive fashion. This bug class can take many forms and led to major incidents such as the DAO hacks[3] in the past.

No reentrancy-related issues have been identified during the review.

2.2.4 DoS Issues

Denial-of-Service attacks are particularly devastating for contracts that maintain any kind of funds for their users, as a successful attack could lead to the funds being ultimately lost for their owners.

No DoS issues have been identified during the review.

2.2.5 Old Solidity Versions

Solidity releases happen rather frequently, often including major changes that could break prior established assumptions. Of course these releases also often fix questionable design decisions such as the following:

In earlier Versions of Solidity, `return` statements in functions having modifiers resulted in the rest of the modifier code not being executed. This behaviour allowed modifiers to deadlock as illustrate by the following modifier:

```
    modifier noReentrancy() {
        require(!locked);
        locked = true;
        _;
        locked = false;
    }
```

This means, that if a function that has this modifier attached returns, the lock will never be released and subsequent calls will always revert.

This behavior was changed in Version 0.4.0[4] and since the analyzed contracts requires version 0.8.0 or greater, it is not affected.

2.2.6 Logic Issues

A major focus of the audit was to detect logic errors that could cause the contract to operate incorrectly and that might lead to e.g., undesired state changes. An example for a logic error might be to give a private function public visibility. The following function initializes the owner of a

3 https://en.wikipedia.org/wiki/The_DAO_(organization)
4 https://github.com/ethereum/solidity/blob/develop/Changelog.md#040-2016-09-08

contract:

```
function initContract() public {
  owner = msg.sender;
}
```

Specifying this function as public enables everyone to become the owner of the contract.

Logic errors are very hard to detect and can become quite involved, when for example using multi-transaction invariants.

During the review, no such issues were identified.

2.3 Project Execution

The project has been executed in the time frame from 2025-02-24 to 2025-02-28 in two person days.

The consultants assigned to this projects were:

- Gregor Kopf

3 Result Overview

An overview of the project results is provided in the following table.

Description	Chapter	Type	Exploitability	Attack Impact
Potential Frontrunning issues in ERC-20	4.1	Observation	Low	Medium

Each identified issue is briefly described by its title, its type, its exploitability and by the impact of a successful exploitation. Technical details for the individual issues are provided in the respective sections of chapter 4 of this document. Details regarding the vulnerability rating scheme used in this document are provided in section 5.

4 Results

The issues identified during the project are described in detail in the following sections. For each finding, there is a technical description, recommended actions and - if necessary and possible - reproduction steps. For details regarding the used vulnerability rating scheme, please refer to section 5 of this document.

4.1 Potential Frontrunning issues in ERC-20

Summary

Type	Location	Exploitability	Attack Impact
Observation	ERC-20 Interface	Low	Medium

Technical Description

The analyzed smart contracts include an ERC-20 contract, which includes the `approve` method for setting an allowance to spend tokens on behalf of another account. It should be noted that this functionality has been criticized in the past, due to potential frontrunning issues.

The core concern can be summarized by the following example: the attacker obtains an allowance to spend N many tokens from the owner's account. After a while, the owner decides they want to reduce this allowance, and hence they issue a transaction to set the allowance to N/2 many tokens. The attacker observes this transaction and now attempts to "sandwich" the owner's transaction into two of their own transactions: the first of which spends N many tokens, the second of which N/2 many tokens. In total, the attacker spent more tokens than the owner originally intended.

It should be noted that there are approaches to mitigate such problems. One option is to first set the allowance to zero, then check if the attacker actually spent any tokens, and then afterwards increase the allowance back to the desired value. This however imposes a certain workload on the owner's side, who now has to perform additional checks.

Recommended Action

This issue mainly serves the purpose of making Taurus aware of the situation. No direct fixes in the code are required, but when dealing with allowances, care should be taken.

Reproduction Steps

This issue has been identified as part of a static analysis, and hence no reproduction steps can be provided.

5 Vulnerability Rating

This section provides a description of the vulnerability rating scheme used in this document. Each finding is rated by its type and its exploitability/impact of a successful exploitation. The meaning of the individual ratings are provided in the following sub-sections.

5.1 Vulnerability Types

Vulnerabilities are rated by the types described in the following table.

Type	Description
Configuration	The finding is a configuration issue
Design	The finding is the result of a design decision
Code	The finding is caused by a coding mistake
Observation	The finding is an observation, which does not necessarily have a direct impact

5.2 Exploitability and Impact

The exploitability of a vulnerability describes the required skill level of an attacker as well as the required resources. Therefore, it provides an indication of the likelihood of exploitation.

Exploitability Rating	Description
Not Exploitable	This finding can most likely not be exploited.
Minimal	Although an attack is theoretically possible, it is extremely unlikely that an attacker will exploit the identified vulnerability.
Low	Exploiting the vulnerability requires the skill-level of an expert. An attack is possible, but difficult pre-conditions (e.g., prior identification and exploitation of other vulnerabilities) exist or the attack requires resources not available to the general public (e.g., expensive equipment). Successful exploitation indicates a dedicated, targeted attack.
Medium	The vulnerability can be exploited under certain pre-conditions (e.g., user interaction or prior authentication). Non-targeted, random attacks are possible for attackers with a medium skill level who perform such attacks on a regular basis.
High	The vulnerability can be exploited immediately without special pre-conditions, by random attackers or in an automated fashion. Only general knowledge about vulnerability exploitation is required.

The following table describes the impact rating used in this document.

Impact Rating	Description
Critical	The vulnerability is a systematic error or it permits compromising the system completely and beyond the scope of the assessment.

Impact Rating	Description
High	The vulnerability permits compromising the systems within the scope completely.
Medium	The vulnerability exceeds certain security rules, but does not lead to a full compromise (e.g., Denial of Service attacks)
Low	The vulnerability has no direct security consequences but provides information which can be used for subsequent attacks.
Informational	The observed finding does not have any direct security consequence; however, addressing the finding can lead to an increase in security or quality of the system in scope.

When rating the impact of a vulnerability, the rating is always performed based on the scope of the analysis. For example, a vulnerability with high impact typically allows an attacker to fully compromise one or all of the core security guarantees of the components in scope. Identical vulnerabilities can therefore be rated differently in different projects.

6 Glossary

Term	Definition
DoS	Denial Of Service