OMNISCIA

May 2, 2024

# SMART CONTRACT AUDIT REPORT

Karpatkey

Token Governor



omniscia.io


info@omniscia.io



Online report: karpatkey-token-governor

Omniscia.io is one of the fastest growing and most trusted blockchain security firms and has rapidly become a true market leader. To date, our team has collectively secured over 370+ clients, detecting 1,500+ high-severity issues in widely adopted smart contracts.

Founded in France at the start of 2020, and with a track record spanning back to 2017, our team has been at the forefront of auditing smart contracts, providing expert analysis and identifying potential vulnerabilities to ensure the highest level of security of popular smart contracts, as well as complex and sophisticated decentralized protocols.

Our clients, ecosystem partners, and backers include leading ecosystem players such as L'Oréal, Polygon, AvaLabs, Gnosis, Morpho, Vesta, Gravita, Olympus DAO, Fetch.ai, and LimitBreak, among others.

To keep up to date with all the latest news and announcements follow us on twitter @omniscia_sec.

omniscia.io


info@omniscia.io

# Token Governor Security Audit

## Audit Report Revisions

| Commit Hash | Date | Audit Report Hash |
| --- | --- | --- |
| 387bdf5b9e | April 22nd 2024 | e50495b298 |
| 12d3a4f883 | May 2nd 2024 | 3ebc5a20bd |

## Audit Overview

We were tasked with performing an audit of the Karpatkey codebase and in particular their Token & Governor module.

Over the course of the audit, we identified certain minor flaws particularly in the way the Governor is configured as well as the way the Token applies its access-control and sanitizes allowlist balance decreases.

We advise the Karpatkey team to closely evaluate all minor findings identified in the report and promptly remediate them as well as consider all optimizational exhibits identified in the report.

## Post-Audit Conclusion

The Karpatkey team iterated through all findings within the report and provided us with a revised commit hash to evaluate all exhibits on.

We evaluated all alleviations performed by Karpatkey and have confirmed that all exhibits have been adequately alleviated or safely acknowledged.

We consider all outputs of the audit report properly consumed by the Karpatkey team rendering this audit engagement concluded.

## Audit Synopsis

| Severity | Identified | Alleviated | Partially Alleviated | Acknowledged |
|---|---|---|---|---|
| Unknown | 1 | 1 | 0 | 0 |
| Informational | 2 | 1 | 0 | 1 |
| Minor | 1 | 1 | 0 | 0 |
| Medium | 1 | 1 | 0 | 0 |
| Major | 0 | 0 | 0 | 0 |

During the audit, we filtered and validated a total of **1 findings utilizing static analysis** tools as well as identified a total of **4 findings during the manual review** of the codebase. We strongly recommend that any minor severity or higher findings are dealt with promptly prior to the project's launch as they can introduce potential misbehaviours of the system as well as exploits.

# Scope

The audit engagement encompassed a specific list of contracts that were present in the commit hash of the repository that was in scope. The tables below detail certain meta-data about the target of the security assessment and a navigation chart is present at the end that links to the relevant findings per file.

## Target

- Repository: **https://github.com/karpatkey/kpk-token**
- Commit: 387bdf5b9e661becf784a6f1f6d94e0e5813e592
- Language: Solidity
- Network: Ethereum
- Revisions: **387bdf5b9e**, **12d3a4f883**

## Contracts Assessed

| File | Total Finding(s) |
| --- | --- |
| **src/contracts/karpatkeyToken.sol (TNE)** | 5 |

# Compilation

The project utilizes `foundry` as its development pipeline tool, containing an array of tests and scripts coded in Solidity.

To compile the project, the `build` command needs to be issued via the `forge` CLI tool:

```bash
forge build
```

The `forge` tool automatically selects Solidity version `0.8.20` based on the version specified within the `foundry.toml` file.

The project contains discrepancies with regards to the Solidity version used as the `pragma` statements of the contracts are open-ended (`^0.8.20`).

We advise them to be locked to `0.8.20` (`=0.8.20`), the same version utilized for our static analysis as well as optimizational review of the codebase.

During compilation with the `foundry` pipeline, no errors were identified that relate to the syntax or bytecode size of the contracts.

# Static Analysis

The execution of our static analysis toolkit identified **5 potential issues** within the codebase of which **4 were ruled out to be false positives** or negligible findings.

The remaining **1 issues** were validated and grouped and formalized into the **1 exhibits** that follow:

| ID | Severity | Addressed | Title |
| --- | --- | --- | --- |
| **TNE-01S** | Medium | Yes | Improper Invocation of EIP-20 `transfer` |

# Manual Review

A **thorough line-by-line review** was conducted on the codebase to identify potential malfunctions and vulnerabilities in Karpatkey's Token & Governor implementation.

As the project at hand implements an EIP-20 token and Governor, intricate care was put into ensuring that the **flow of funds within the system conforms to the specifications and restrictions** laid forth within the protocol's specification.

We validated that **all state transitions of the system occur within sane criteria** and that all rudimentary formulas within the system execute as expected. We **pinpointed certain minor vulnerabilities** within the system which could have had **moderate ramifications** to its overall operation **under specific unlikely scenarios**.

Additionally, the system was investigated for any other commonly present attack vectors such as re-entrancy attacks, mathematical truncations, logical flaws and **ERC / EIP** standard inconsistencies. The documentation of the project was satisfactory to the extent it need be.

A total of **4 findings** were identified over the course of the manual review of which **2 findings** concerned the behaviour and security of the system. The non-security related findings, such as optimizations, are included in the separate **Code Style** chapter.

The finding table below enumerates all these security / behavioural findings:

| ID | Severity | Addressed | Title |
|---|---|---|---|
| **TNE-01M** | Unknown | Nullified | Potentially Incorrect Application of Transfer Allowlist Restriction |
| **TNE-02M** | Minor | Yes | Potentially Dangerous Restriction of Allowance Reduction |

# Code Style

During the manual portion of the audit, we identified **2 optimizations** that can be applied to the codebase that will decrease the operational cost associated with the execution of a particular function and generally ensure that the project complies with the latest best practices and standards in Solidity.

Additionally, this section of the audit contains any opinionated adjustments we believe the code should make to make it more legible as well as truer to its purpose.

These optimizations are enumerated below:

| ID | Severity | Addressed | Title |
| --- | --- | --- | --- |
| **TNE-01C** | Informational | Acknowledged | Generic Typographic Mistake |
| **TNE-02C** | Informational | Yes | Ineffectual Usage of Safe Arithmetics |

# karpatkeyToken Static Analysis Findings

## TNE-01S: Improper Invocation of EIP-20 `transfer`

| Type | Severity | Location |
| --- | --- | --- |
| Standard Conformity | Medium | karpatkeyToken.sol:L252 |

### Description:

The linked statement does not properly validate the returned `bool` of the **EIP-20** standard `transfer` function. As the **standard dictates**, callers **must not** assume that `false` is never returned.

### Impact:

If the code mandates that the returned `bool` is `true`, this will cause incompatibility with tokens such as USDT / Tether as no such `bool` is returned to be evaluated causing the check to fail at all times. On the other hand, if the token utilized can return a `false` value under certain conditions but the code does not validate it, the contract itself can be compromised as having received / sent funds that it never did.

### Example:

src/contracts/karpatkeyToken.sol

SOL

```sol
252 token.transfer(recipient, value);
```

### Recommendation:

Since not all standardized tokens are **EIP-20** compliant (such as Tether / USDT), we advise a safe wrapper library to be utilized instead such as `SafeERC20` by OpenZeppelin to opportunistically validate the returned `bool` only if it exists.

### Alleviation:

The `SafeERC20` library of the OpenZeppelin dependency is now properly imported to the codebase and its `SafeERC20::safeTransfer` function is correctly invoked in place of the potentially unhandled **EIP-20** `ERC20::transfer` invocation, addressing this exhibit.

# karpatkeyToken Manual Review Findings

## TNE-01M: Potentially Incorrect Application of Transfer Allowlist Restriction

| Type | Severity | Location |
| --- | --- | --- |
| Logical Fault | Unknown | karpatkeyToken.sol:L329 |

### Description:

The `karpatkeyToken::_update` function that has been overridden will impose several access control checks when the contract is paused, including whether the `from` address is transfer-allowlisted.

We consider this incorrect given that the current mechanism permits a `_transferAllowlisted` member to approve others to execute `ERC20Upgradeable::transferFrom` operations on their behalf effectively bypassing the allowlist.

### Impact:

The severity of this exhibit will be adjusted depending on whether the Karpatkey team ultimately considers the current behaviour to go against their business requirements.

### Example:

src/contracts/karpatkeyToken.sol

SOL

```
312 /**
313  * @dev Transfers a `value` amount of tokens from `from` to `to`, or
alternatively mints (or burns) if `from`
314  * (or `to`) is the zero address. If the contract is paused and the caller is not
the contract's owner, and
315  * additionally if `from` is not the contract's owner and `from` is not
allowlisted to transfer tokens, then
316  * `value` is spent from the transfer allowance for `from` to `to`.
317  * Reverts with {TransferToTokenContract} if `to` is the token contract itself.
318  * See {ERC20Upgradeable-_update}
319  * @param from Address to transfer tokens from.
320  * @param to Address to transfer tokens to.
321  * @param value Amount of tokens to be transferred.
```

## Example (Cont.):

SOL

```sol
 */
function _update(address from, address to, uint256 value) internal
override(ERC20Upgradeable, ERC20VotesUpgradeable) {
  if (to == address(this)) {
    revert TransferToTokenContract();
  }
  // 'from != owner()' is included to avoid involving the transfer allowance
mechanism when tokens are
  // transferred from the owner via {transferFrom}
  if (paused() && _msgSender() != owner() && from != owner() &&
!_transferAllowlisted[from]) {
    _spendTransferAllowance(from, to, value);
  }
  super._update(from, to, value);
}
```

### Recommendation:

We advise the `ContextUpgradeable::_msgSender` to be validated instead, ensuring that the caller is allowlisted rather than the `from` member the transaction extracts funds from.

### Alleviation:

The Karpatkey team evaluated this exhibit and has clarified that the exhibit's described behaviour aligns with the business requirements of the Karpatkey team.

As such, we consider this exhibit inapplicable as it pertains to desirable behaviour.

# TNE-02M: Potentially Dangerous Restriction of Allowance Reduction

| Type | Severity | Location |
| --- | --- | --- |
| Logical Fault | Minor | karpatkeyToken.sol:L217-L219 |

### Description:

The current `karpatkeyToken::decreaseTransferAllowance` mechanism suffers from a well-known **EIP-20** race-condition whereby a user may detect that their transfer allowance is being reduced to `0` and transfer a single `wei` to prevent the allowance reduction from ever succeeding.

### Impact:

The present `karpatkeyToken::decreaseTransferAllowance` mechanism can be hijacked to prevent execution by continuously mutating the existing transfer allowance below the reduced value.

### Example:

src/contracts/karpatkeyToken.sol

SOL

```
196 /**
197  * @notice Decreases the transfer allowance for an account to transfer tokens to
a specified recipient
198  * when the contract is paused.
199  * @dev Decreases the transfer allowance for `sender` and `recipient` by
`subtractedValue`. Can only be called by the
200  * token contract's owner.
201  * Reverts with {DecreasedTransferAllowanceBelowZero} if the transfer allowance
would be decreased below zero.
202  * Reverts with {TransferApprovalWhenUnpaused} if called when the contract is
unpaused.
203  * See {_approveTransfer}.
204  * @param sender Address that may be allowed to transfer tokens.
205  * @param recipient Address to which tokens may be transferred.
```

## Example (Cont.):

```sol
 * @param subtractedValue Amount to decrease the transfer allowance by.
 */
function decreaseTransferAllowance(
  address sender,
  address recipient,
  uint256 subtractedValue
) public onlyOwner returns (bool success) {
  if (!paused()) {
    revert TransferApprovalWhenUnpaused();
  }
  uint256 currentTransferAllowance = _transferAllowances[sender][recipient];
  if (currentTransferAllowance < subtractedValue) {
    revert DecreasedTransferAllowanceBelowZero(sender, recipient, currentTransferAllowance, subtractedValue);
  }
  _approveTransfer(sender, recipient, currentTransferAllowance - subtractedValue);
  return true;
}
```

### Recommendation:

We advise the code to reduce the allowance by the minimum between `subtractedValue` and `currentTransferAllowance` instead, ensuring that an allowance reduction will go through regardless of whether the allowance has mutated between a decrease operation's submission and its execution in the network.

### Alleviation:

The code was updated per our recommendation, setting the approval to `0` if the `subtractedValue` exceeds the `currentTransferAllowance` to prevent race-condition exploitation scenarios from manifesting.

# karpatkeyToken Code Style Findings

## TNE-01C: Generic Typographic Mistake

| Type | Severity | Location |
| --- | --- | --- |
| Code Style | Informational | karpatkeyToken.sol:L24 |

### Description:

The referenced line contains a typographical mistake (i.e. `private` variable without an underscore prefix) or generic documentational error (i.e. copy-paste) that should be corrected.

### Example:

src/contracts/karpatkeyToken.sol

SOL

```sol
24  contract karpatkeyToken is
```

### Recommendation:

We advise this to be done so to enhance the legibility of the codebase.

### Alleviation:

The Karpatkey team clarified that the `karpatkey` brand is meant to be represented with a lowercase `k`, and has opted to retain the current naming convention.

# TNE-02C: Ineffectual Usage of Safe Arithmetics

| Type | Severity | Location |
| --- | --- | --- |
| Language Specific | Informational | karpatkeyToken.sol:L220 |

### Description:

The linked mathematical operation is guaranteed to be performed safely by surrounding conditionals evaluated in either `require` checks or `if-else` constructs.

### Example:

src/contracts/karpatkeyToken.sol

SOL

```sol
217 if (currentTransferAllowance < subtractedValue) {
218   revert DecreasedTransferAllowanceBelowZero(sender, recipient,
currentTransferAllowance, subtractedValue);
219 }
220 _approveTransfer(sender, recipient, currentTransferAllowance - subtractedValue);
```

### Recommendation:

Given that safe arithmetics are toggled on by default in `pragma` versions of `0.8.X`, we advise the linked statement to be wrapped in an `unchecked` code block thereby optimizing its execution cost.

### Alleviation:

The referenced subtraction has been wrapped in an `unchecked` code block as advised, optimizing its gas cost.

# Finding Types

A description of each finding type included in the report can be found below and is linked by each respective finding. A full list of finding types Omniscia has defined will be viewable at the central audit methodology we will publish soon.

## Input Sanitization

As there are no inherent guarantees to the inputs a function accepts, a set of guards should always be in place to sanitize the values passed in to a particular function.

## Indeterminate Code

These types of issues arise when a linked code segment may not behave as expected, either due to mistyped code, convoluted if blocks, overlapping functions / variable names and other ambiguous statements.

## Language Specific

Language specific issues arise from certain peculiarities that the Circom language boasts that discerns it from other conventional programming languages.

## Curve Specific

Circom defaults to using the BN128 scalar field (a 254-bit prime field), but it also supports BSL12-381 (which has a 255-bit scalar field) and Goldilocks (with a 64-bit scalar field). However, since there are no constants denoting either the prime or the prime size in bits available in the Circom language, some Circomlib templates like `Sign` (which returns the sign of the input signal), and `AliasCheck` (used by the strict versions of `Num2Bits` and `Bits2Num`), hardcode either the BN128 prime size or some other constant related to BN128. Using these circuits with a custom prime may thus lead to unexpected results and should be avoided.

## Code Style

In these types of findings, we identify whether a project conforms to a particular naming convention and whether that convention is consistent within the codebase and legible. In case of inconsistencies, we point them out under this category. Additionally, variable shadowing falls under this category as well which is identified when a local-level variable contains the same name as a toplevel variable in the circuit.

## Mathematical Operations

This category is used when a mathematical issue is identified. This implies an issue with the implementation of a calculation compared to the specifications.

## Logical Fault

This category is a bit broad and is meant to cover implementations that contain flaws in the way they are implemented, either due to unimplemented functionality, unaccounted-for edge cases or similar extraordinary scenarios.

## Privacy Concern

This category is used when information that is meant to be kept private is made public in some way.

## Proof Concern

Under-constrained signals are one of the most common issues in zero-knowledge circuits. Issues with proof generation fall under this category.

# Severity Definition

In the ever-evolving world of blockchain technology, vulnerabilities continue to take on new forms and arise as more innovative projects manifest, new blockchain-level features are introduced, and novel layer-2 solutions are launched. When performing security reviews, we are tasked with classifying the various types of vulnerabilities we identify into subcategories to better aid our readers in understanding their impact.

Within this page, we will clarify what each severity level stands for and our approach in categorizing the findings we pinpoint in our audits. To note, all severity assessments are performed **as if the contract's logic cannot be upgraded** regardless of the underlying implementation.

## Severity Levels

There are five distinct severity levels within our reports; `unknown`, `informational`, `minor`, `medium`, and `major`. A TL;DR overview table can be found below as well as a dedicated chapter to each severity level:

| | Impact (None) | Impact (Low) | Impact (Moderate) | Impact (High) |
|---|---|---|---|---|
| **Likelihood (None)** | Informational | Informational | Informational | Informational |
| **Likelihood (Low)** | Informational | Minor | Minor | Medium |
| **Likelihood (Moderate)** | Informational | Minor | Medium | Major |
| **Likelihood (High)** | Informational | Medium | Major | Major |

### Unknown Severity

The `unknown` severity level is reserved for misbehaviors we observe in the codebase that cannot be quantified using the above metrics. Examples of such vulnerabilities include potentially desirable system behavior that is undocumented, reliance on external dependencies that are out-of-scope but could result in some form of vulnerability arising, use of external out-of-scope contracts that appears incorrect but cannot be pinpointed, and other such vulnerabilities.

In general, `unknown` severity level vulnerabilities require follow-up information by the project being audited and are either adjusted in severity (if valid), or marked as nullified (if invalid).

Additionally, the `unknown` severity level is sometimes assigned to centralization issues that cannot be assessed in likelihood due to their exploitation being tied to the honesty of the project's team.

### Informational Severity

The `informational` severity level is dedicated to findings that do not affect the code functionally and tend to be stylistic or optimizational in nature. Certain edge cases are also set under `informational` vulnerabilities, such as overflow operations that will not manifest in the lifetime of the contract but should be guarded against as a best practice, to give an example.

### Minor Severity

The `minor` severity level is meant for vulnerabilities that require functional changes in the code but tend to either have little impact or be unlikely to be recreated in a production environment. These findings can be acknowledged except for findings with a moderate impact but low likelihood which must be alleviated.

### Medium Severity

The `medium` severity level is assigned to vulnerabilities that must be alleviated and have an observable impact on the overall project. These findings can only be acknowdged if the project deems them desirable behavior and we disagree with their point-of-view, instead urging them to reconsider their stance while marking the exhibit as acknowledged given that the project has ultimate say as to what vulnerabilities they end up patching in their system.

### Major Severity

The `major` severity level is the maximum that can be specified for a finding and indicates a significant flaw in the code that must be alleviated.

## Likelihood & Impact Assessment

As the preface chapter specifies, the blockchain space is constantly reinventing itself meaning that new vulnerabilities take place and our understanding of what security means differs year-to-year.

In order to reliably assess the likelihood and impact of a particular vulnerability, we instead apply an abstract measurement of a vulnerability's impact, duration the impact is applied for, and probability that the vulnerability would be exploited in a production environment.

Our proposed definitions are inspired by multiple sources in the security community and are as follows:

# Disclaimer

The following disclaimer applies to all versions of the audit report produced (preliminary / public / private) and is in effect for all past, current, and future audit reports that are produced and hosted under Omniscia:

## IMPORTANT TERMS & CONDITIONS REGARDING OUR SECURITY AUDITS/REVIEWS/REPORTS AND ALL PUBLIC/PRIVATE CONTENT/DELIVERABLES

Omniscia ("Omniscia") has conducted an independent security review to verify the integrity of and highlight any vulnerabilities, bugs or errors, intentional or unintentional, that may be present in the codebase that were provided for the scope of this Engagement.

Blockchain technology and the cryptographic assets it supports are nascent technologies. This makes them extremely volatile assets. Any assessment report obtained on such volatile and nascent assets may include unpredictable results which may lead to positive or negative outcomes.

In some cases, services provided may be reliant on a variety of third parties. This security review does not constitute endorsement, agreement or acceptance for the Project and technology that was reviewed. Users relying on this security review should not consider this as having any merit for financial advice or technological due diligence in any shape, form or nature.

The veracity and accuracy of the findings presented in this report relate solely to the proficiency, competence, aptitude and discretion of our auditors. Omniscia and its employees make no guarantees, nor assurance that the contracts are free of exploits, bugs, vulnerabilities, deprecation of technologies or any system / economical / mathematical malfunction.

This audit report shall not be printed, saved, disclosed nor transmitted to any persons or parties on any objective, goal or justification without due written assent, acquiescence or approval by Omniscia.

All the information/opinions/suggestions provided in this report does not constitute financial or investment advice, nor should it be used to signal that any person reading this report should invest their funds without sufficient individual due diligence regardless of the findings presented in this report.

Information in this report is provided 'as is'. Omniscia is under no covenant to the completeness, accuracy or solidity of the contracts reviewed. Omniscia's goal is to help reduce the attack vectors/surface and the high level of variance associated with utilizing new and consistently changing technologies.

Omniscia in no way claims any guarantee, warranty or assurance of security or functionality of the technology that was in scope for this security review.

In no event will Omniscia, its partners, employees, agents or any parties related to the design/creation of this security review be ever liable to any parties for, or lack thereof, decisions and/or actions with regards to the information provided in this security review.

Cryptocurrencies and all other technologies directly or indirectly related to cryptocurrencies are not standardized, highly prone to malfunction and extremely speculative by nature. No due diligence and/or safeguards may be insufficient and users should exercise maximum caution when participating and/or investing in this nascent industry.

The preparation of this security review has made all reasonable attempts to provide clear and actionable recommendations to the Project team (the “client”) with respect to the rectification, amendment and/or revision of any highlighted issues, vulnerabilities or exploits within the contracts in scope for this engagement.

It is the sole responsibility of the Project team to provide adequate levels of test and perform the necessary checks to ensure that the contracts are functioning as intended, and more specifically to ensure that the functions contained within the contracts in scope have the desired intended effects, functionalities and outcomes, as documented by the Project team.

All services, the security reports, discussions, work product, attack vectors description or any other materials, products or results of this security review engagement is provided "as is" and "as available" and with all faults, uncertainty and defects without warranty or guarantee of any kind.

Omniscia will assume no liability or responsibility for delays, errors, mistakes, or any inaccuracies of content, suggestions, materials or for any loss, delay, damage of any kind which arose as a result of this engagement/security review.

Omniscia will assume no liability or responsibility for any personal injury, property damage, of any kind whatsoever that resulted in this engagement and the customer having access to or use of the products, engineers, services, security report, or any other other materials.

For avoidance of doubt, this report, its content, access, and/or usage thereof, including any associated services or materials, shall not be considered or relied upon as any form of financial, investment, tax, legal, regulatory, or any other type of advice.