



2023

SMART CONTRACT SECURITY ANALYSIS

PREPARED BY
Saulidity

PRESENTED TO
Giant Mammoth



2023
SAULIDITY

SECURITY REPORT



Smart Contract
Audit



saulidity.com
Saulidity
@Saulidity

DISCLAIMER

This report does not constitute financial advice, and Saulidity is not accountable or liable for any negative consequences resulting from this report, nor may Saulidity be held liable in any way. You agree to the terms of this disclaimer by reading any part of the report. If you do not agree to the terms, please stop reading this report immediately and delete and destroy any and all copies of this report that you have downloaded and/or printed. This report was entirely based on information given by the audited party and facts that existed prior to the audit. Saulidity and/or its auditors cannot be held liable for any outcome, including modifications (if any) made to the contract(s) for the audit that was completed. No modifications have been made to the contract(s) by the Saulidity team unless it is indicated explicitly. The audit does not include the project team, website, logic, or tokenomics, but if it does, it will be indicated explicitly. The security is evaluated only on the basis of smart contracts only. There were no security checks performed on any apps or activities. There has not been a review of any product codes. It is assumed by Saulidity that the information and materials given were not tampered with, censored, or misrepresented. Even if this report exists and Saulidity makes every effort to uncover any security flaws, you should not rely completely on it and should conduct your own independent research. Saulidity hereby excludes all liability and responsibility, and neither you nor any other person shall have any claim against Saulidity, for any amount or kind of loss or damage that may result to you or any other person or any kind of company, community, association and institution. Saulidity is the exclusive owner of this report, and it is published by Saulidity. Without Saulidity's express written authorization, use of this report for any reason other than a security interest in the individual contacts, or use of sections of this report, is forbidden.

Table of Contents

- 02 Saulidity**
- 03 Introduction**
- 04 Scope**
- 05 Appendix**
- 06 SC Weakness Registry**
- 09 Audit & Project Information**
- 10 Summary Table**
- 11 Executive Summary**
- 12 Inheritance**
- 22 Call Graph**
- 32 Contract Interaction**
- 42 Analysis**
- 46 Testing Standards**

Saulidity

Saulidity is a renowned cybersecurity firm specializing in the analysis and development of Smart contracts. Saulidity, as a full-service security organization, can help with a variety of audits and project development.

In a market where confidence and trust are key, a genuine project may simply increase its user base enormously with an official audit performed by Saulidity.

Introduction

For a thorough understanding of the audit, please read the entire document.

The goal of the audit was to find any potential smart contract security problems and vulnerabilities.

The information in this report should be used to understand the smart contract's risk exposure and as a guide to improving the smart contract's security posture by addressing the concerns that were discovered.

During our audit, we conducted a thorough inquiry using automated analysis and manual review approaches.

The security specialists did a complete study independently of one another in order to uncover any security issues in the contracts as comprehensively as feasible. For optimum security and professionalism, all of our audits are undertaken by at least two independent auditors.

The project's website, logic, or tokenomics have not been vetted by the Saulidity team.

Scope

We analyze smart contracts for both well-known and more specific vulnerabilities.

Here are some of the most well-known vulnerabilities that are taken into account but not limited to:

- Reentrancy
- Timestamp Dependence
- Gas Limit and Loops
- DoS with (Unexpected) Throw
- DoS with Block Gas Limit
- Transaction-Ordering Dependence
- Style guide violation
- Transfer forwards all gas
- API violation
- Compiler version not fixed
- Unchecked external call - Unchecked math
- Unsafe type inference
- Implicit visibility level

Appendix

Vulnerabilities can be divided into four threat levels: Critical, High, Medium and Low. The classification is mainly based on the impact, likelihood of utilization and other factors.

Critical flaws can result in the loss of assets or the alteration of data and are often simple to exploit.

High-level vulnerabilities are challenging to exploit, but they can have a big influence on how smart contracts are executed, such as giving the public access to key features.

Although medium-level vulnerabilities should be fixed, they generally cannot result in the loss of assets or the manipulation of data.

Low-level flaws are typically caused by code fragments that are out-of-date, useless, etc. and cannot significantly affect execution.

SC Weakness Registry

ITEM	DESCRIPTION
Default Visibility	Functions and state variables visibility should be set explicitly. Visibility levels should be specified consciously.
Integer Overflow and Underflow	If unchecked math is used, all math operations should be safe from overflows and underflows.
Outdated Compiler Version	It is recommended to use a recent version of the Solidity compiler.
Floating Pragma	Contracts should be deployed with the same compiler version and flags that they have been tested thoroughly.
Unchecked Call Return Value	The return value of a message call should be checked.
Access Control & Authorization	Ownership takeover should not be possible. All crucial functions should be protected. Users could not affect data that belongs to other users.
Selfdestruct	The contract should not be destroyed until it has funds belonging to users.
Check-Effect-Interaction	CEI pattern should be followed if the code performs any external call.

SC Weakness Registry

ITEM	DESCRIPTION
Uninitialized Storage Pointer	Storage type should be set explicitly if the compiler version is < 0.5.0.
Assert Violation	Properly functioning code should never reach a failing assert statement.
Deprecated Solidity Functions	Deprecated built-in functions should never be used.
Delegatecall to Untrusted Callee	Delegatecalls should only be allowed to trusted addresses.
Denial of Service	Execution of the code should never be blocked by a specific contract state unless it is required.
Race Conditions	Race Conditions and Transactions Order Dependency should not be possible.
Authorization through tx.origin	tx.origin should not be used for authorization.
Block values as a proxy for time	Block numbers should not be used for time calculations.

SC Weakness Registry

ITEM	DESCRIPTION
Signature Unique Id	Signed messages should always have a unique id. A transaction hash should not be used as a unique id.
Shadowing State Variable	State variables should not be shadowed.
Weak Sources of Randomness	Random values should never be generated from Chain Attributes.
Incorrect Inheritance Order	When inheriting multiple contracts, especially if they have identical functions, a developer should carefully specify inheritance in the correct order.
Calls Only to Trusted Addresses	All external calls should be performed only to trusted addresses.
Presence of unused variables	The code should not contain unused variables if this is not justified by design.

Audit & Project Information

	Project Name	Giant Mammoth
	Contract Information	https://github.com/MammothDevMaster/contracts/
	Report ID	mSAUL001 1.0
	Website	https://www.mmtchain.io
	Contact	contact@mmtchain.io
	Contact Information	SamuelNg@mmtchain.io
	Code language	Solidity

Summary Table

SEVERITY	FOUND
Critical	0
High	0
Medium	0
Low	0
Lowest / Code Style / Optimized Practice	4

Executive Summary

ACCORDING TO THE ANALYSIS, **THERE ARE NO CRITICAL SEVERITY SECURITY VULNERABILITIES.**

ALL ISSUES FOUND DURING ANALYSIS WERE REVIEWED, AND FALSE POSITIVES WERE ELIMINATED. THE FINDINGS ARE PRESENTED IN THE ANALYSIS SECTION OF THE REPORT.

Inheritance

Issue: Inheritance

Severity: -

Location: General

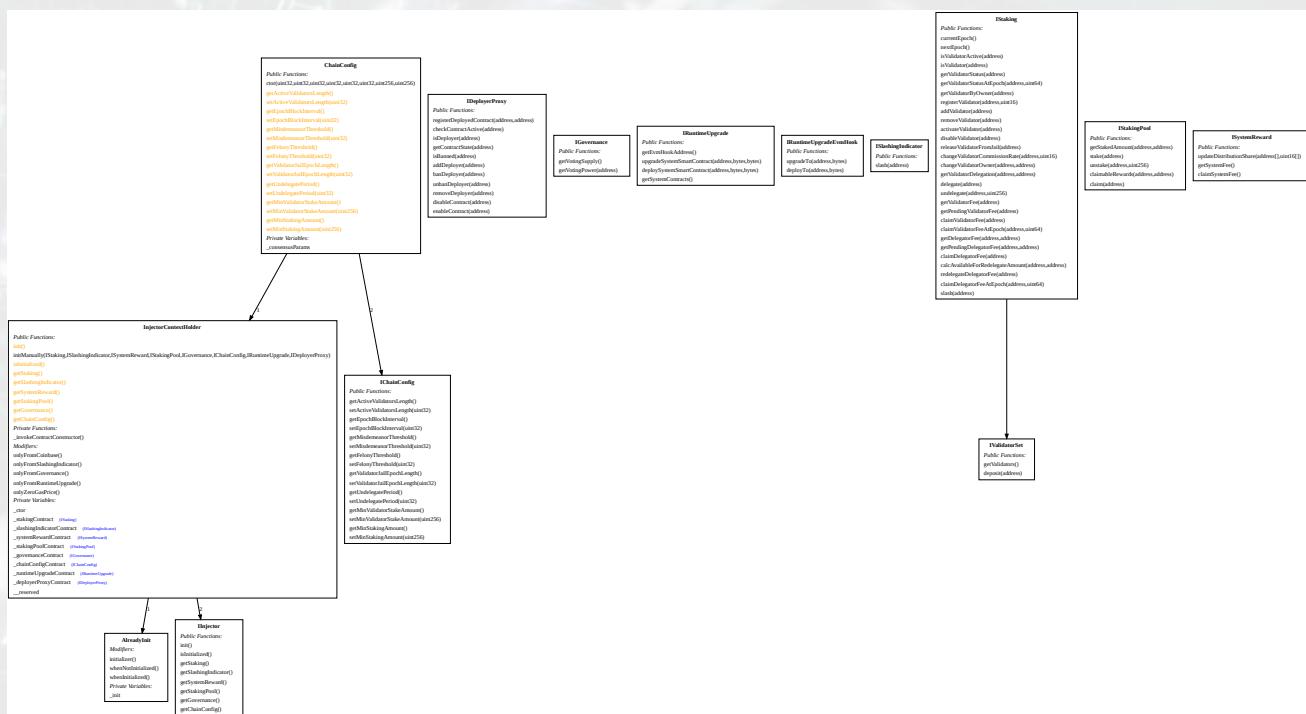
Description: Inheritance is a fundamental concept in object-oriented programming (OOP) that allows a class (referred to as a child or derived class) to inherit characteristics and functionalities from another class (known as a parent or base class).

In the context of smart contracts in Solidity, inheritance is used to establish relationships between contracts, enabling code reuse, responsibility separation, and promoting modularity.

In a smart contract, we can observe inheritance among the different contracts:

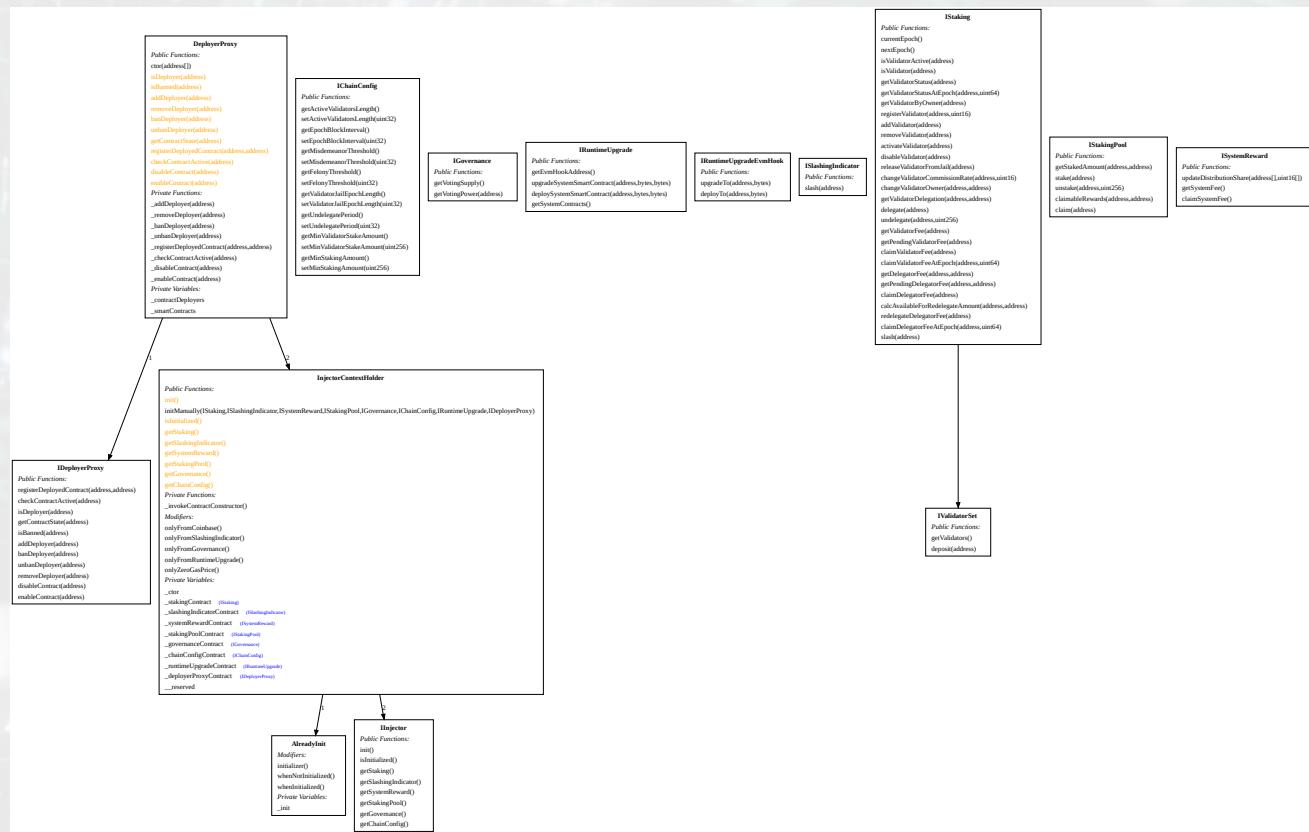
Inheritance

ChainConfig.sol



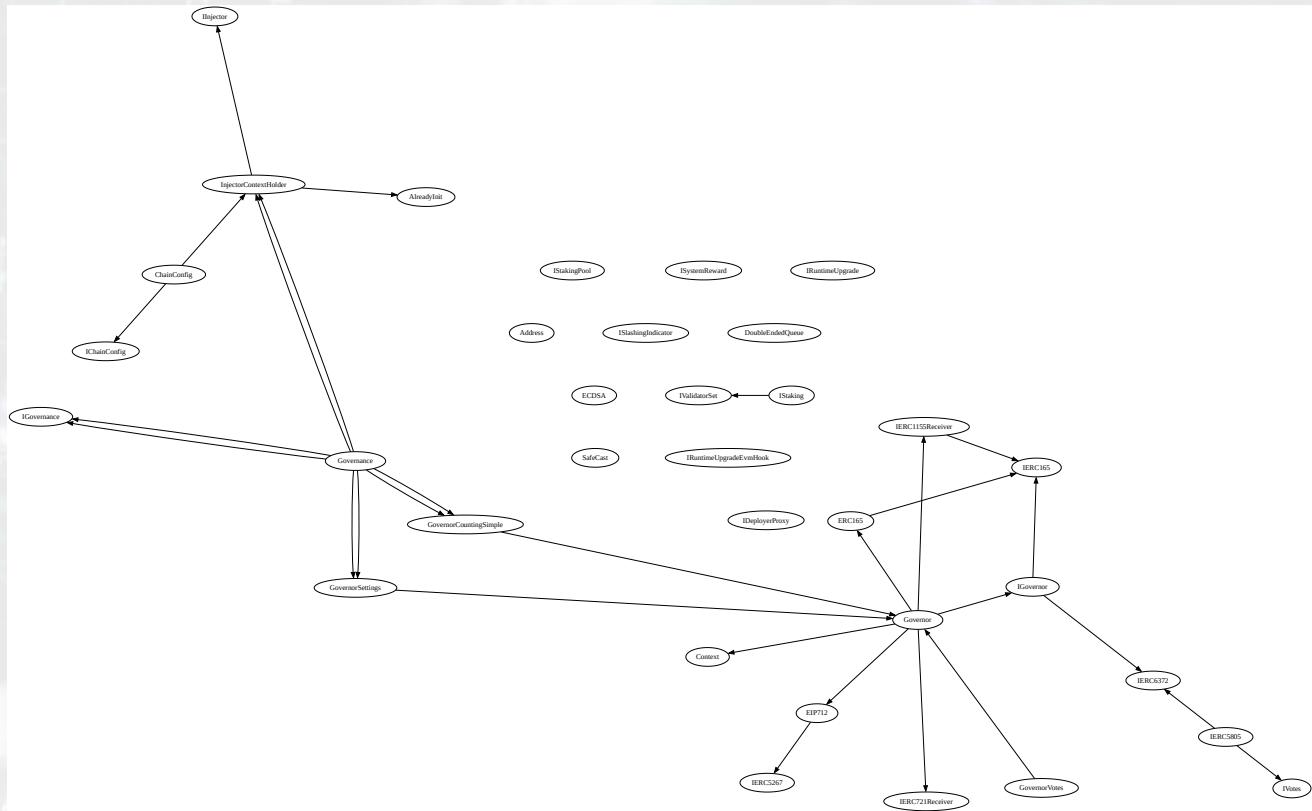
Inheritance

DeployerProxy.sol



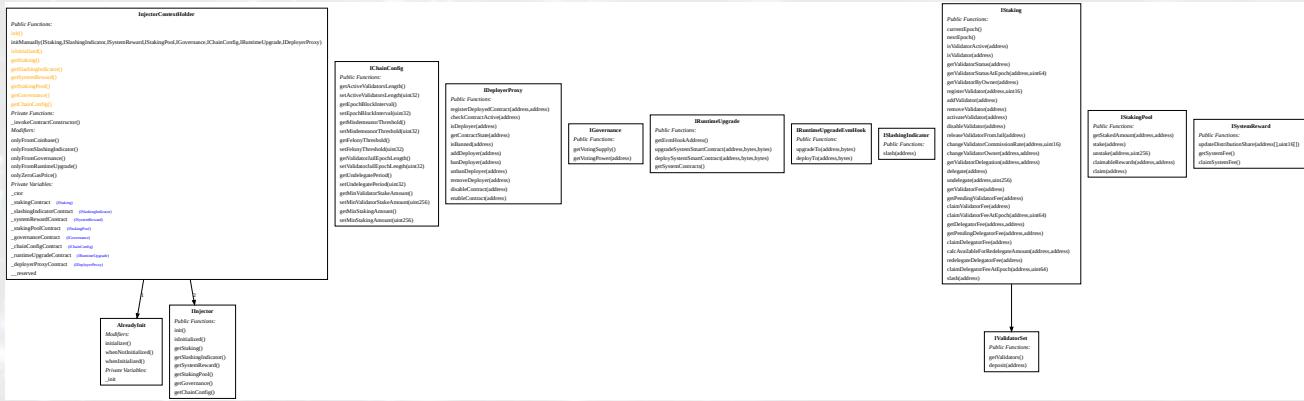
Inheritance

Gouvernance.sol



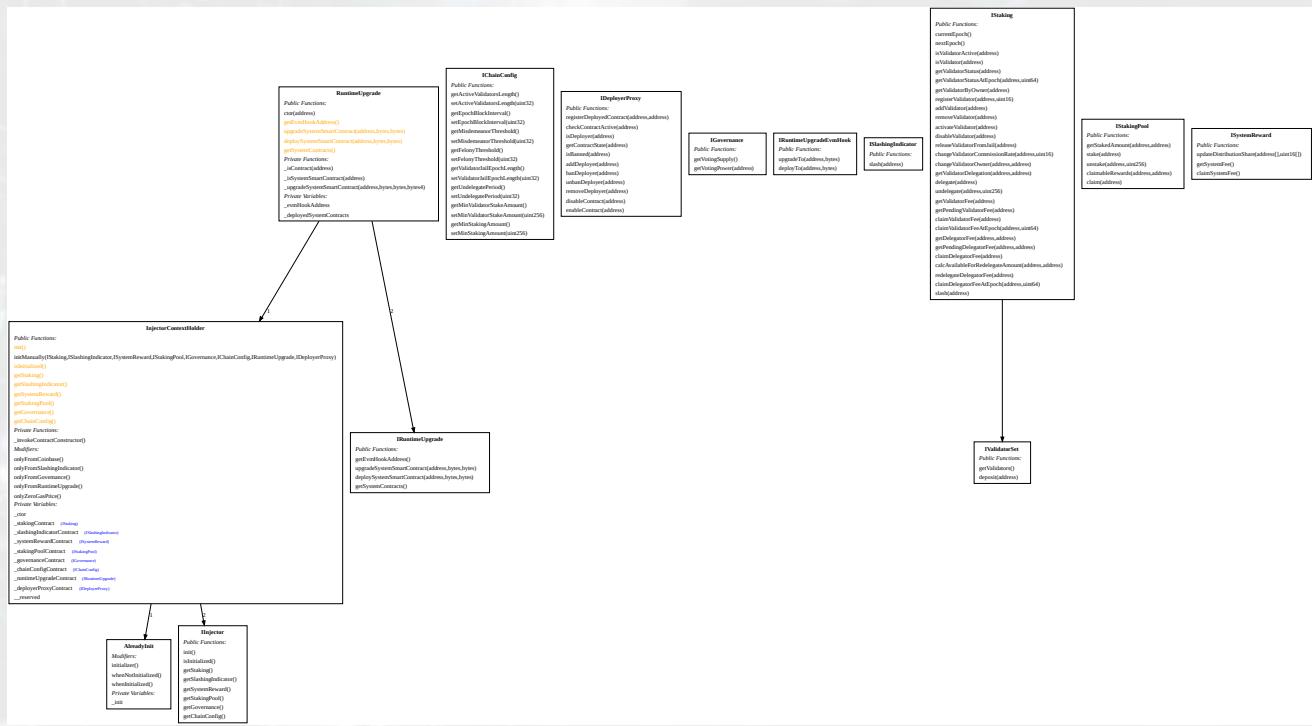
Inheritance

Injector.sol



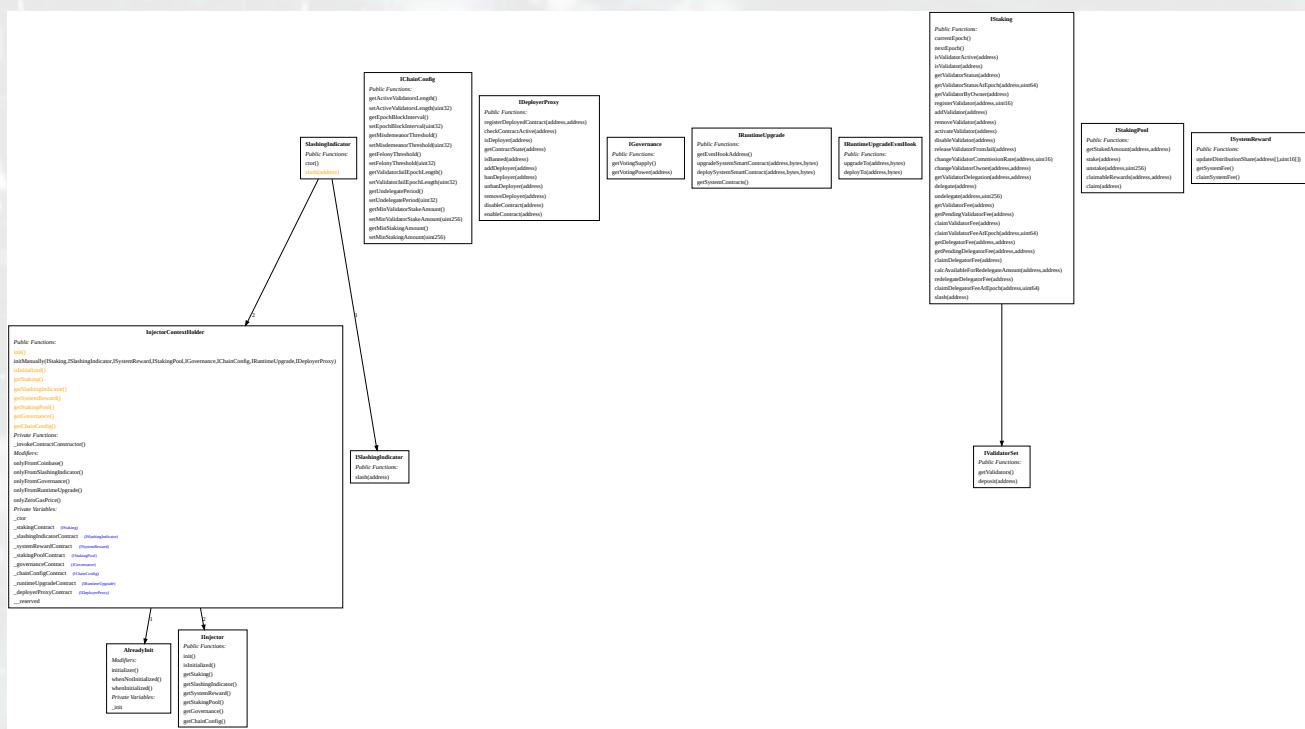
Inheritance

RuntimeUpgrade.sol



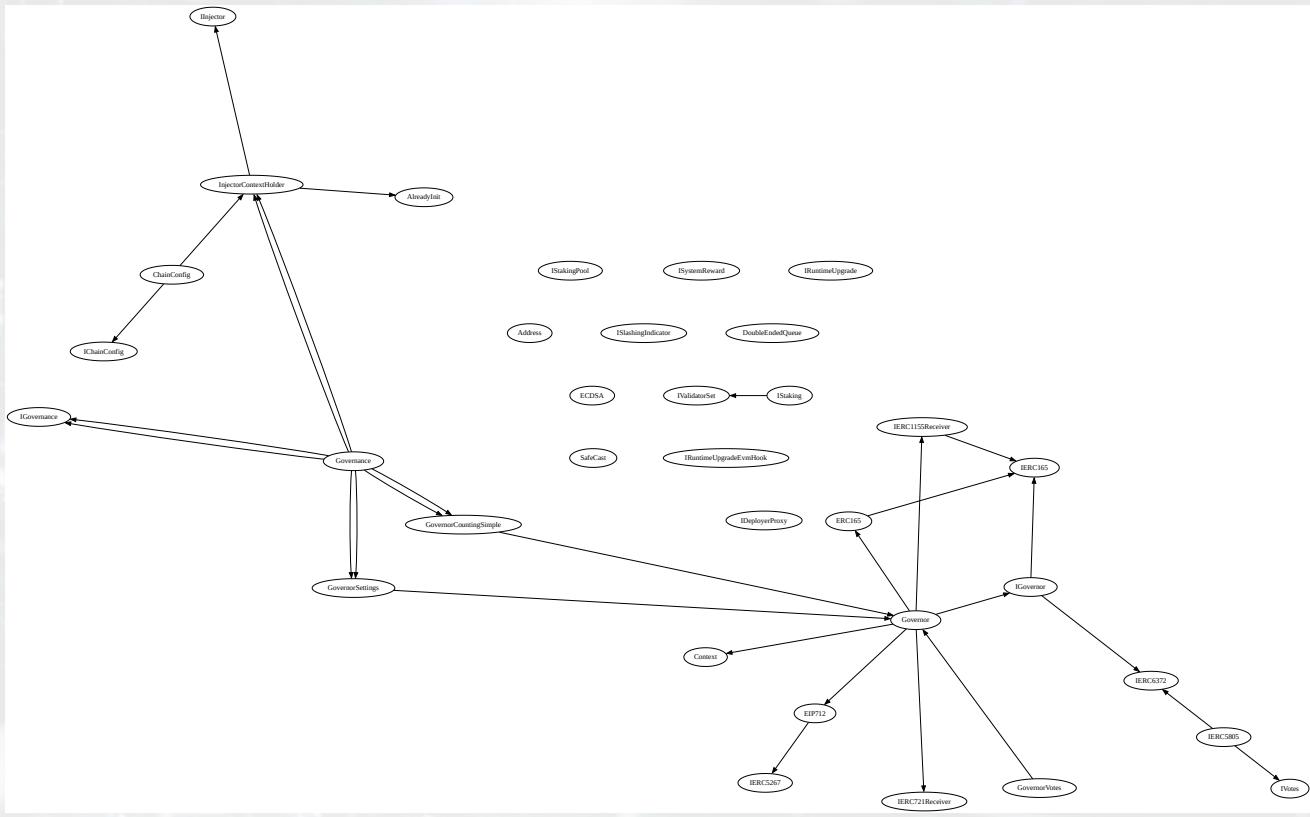
Inheritance

SlashingIndicator.sol



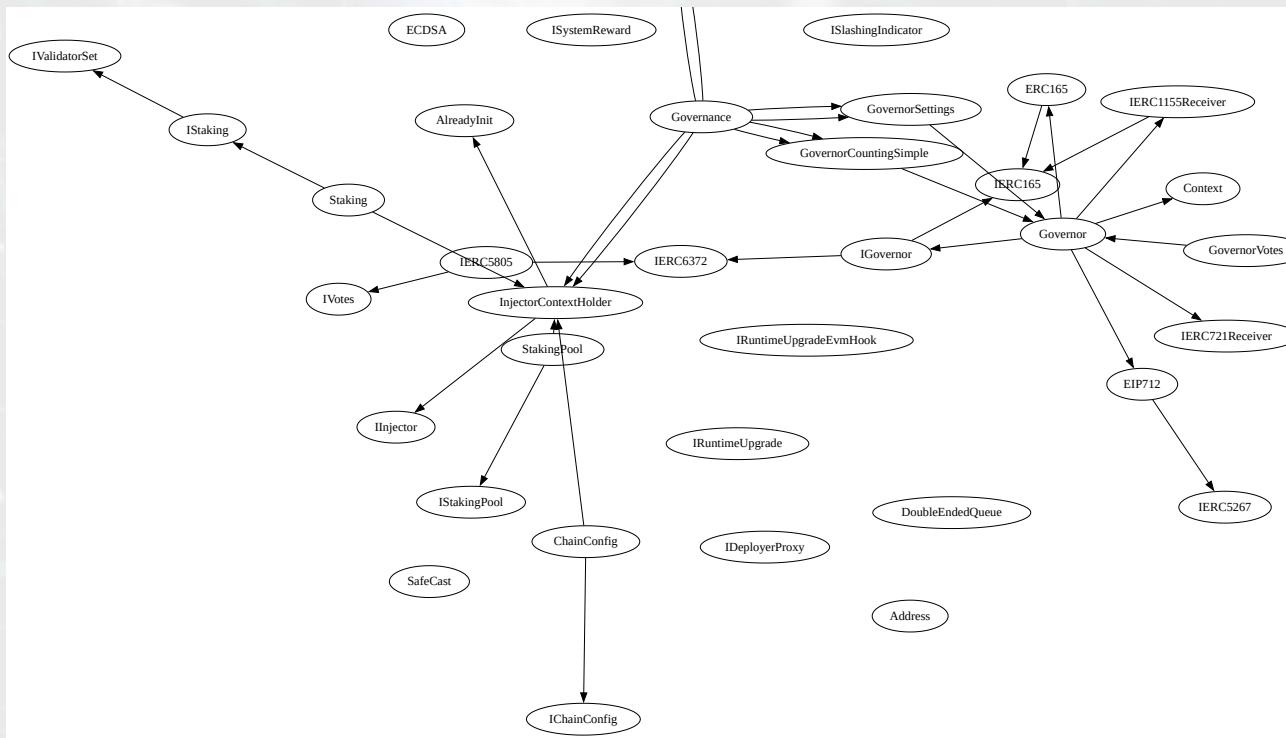
Inheritance

Staking.sol



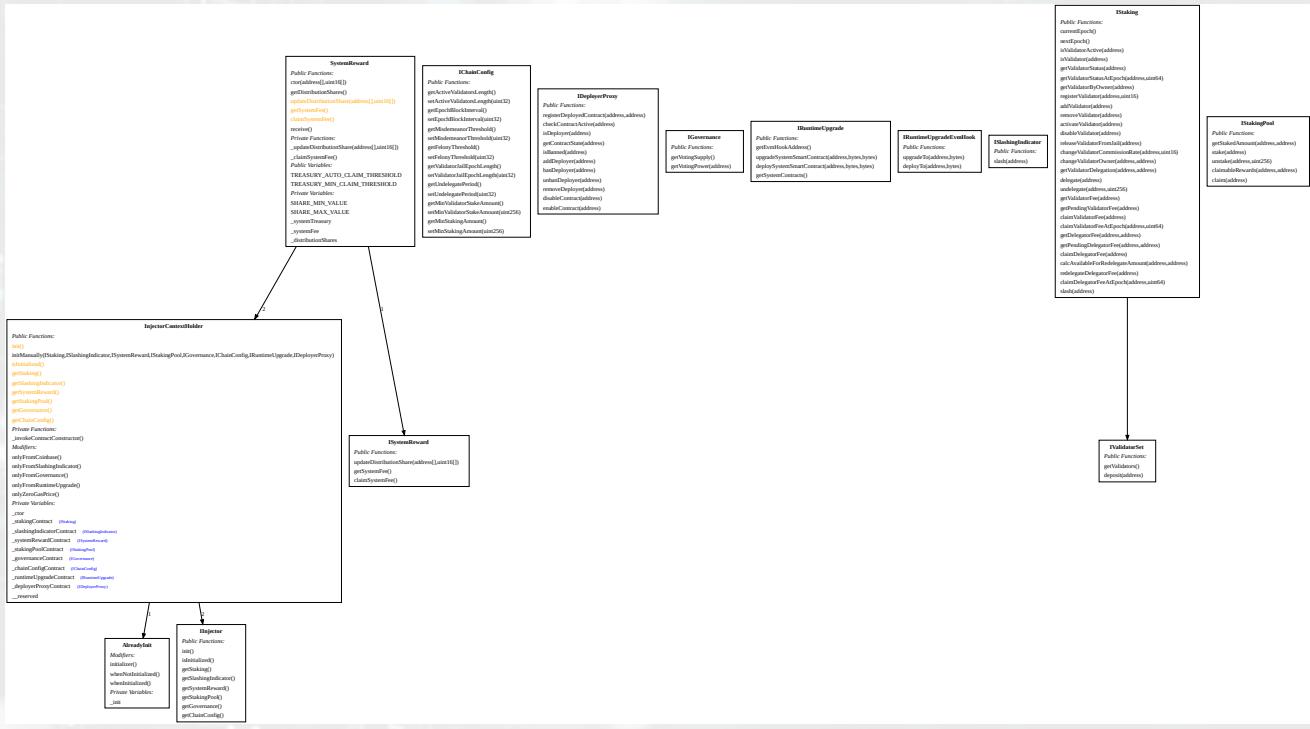
Inheritance

StakingPool.sol



Inheritance

SystemReward.sol



Call Graph

Issue: Call Graph

Severity: -

Location: General

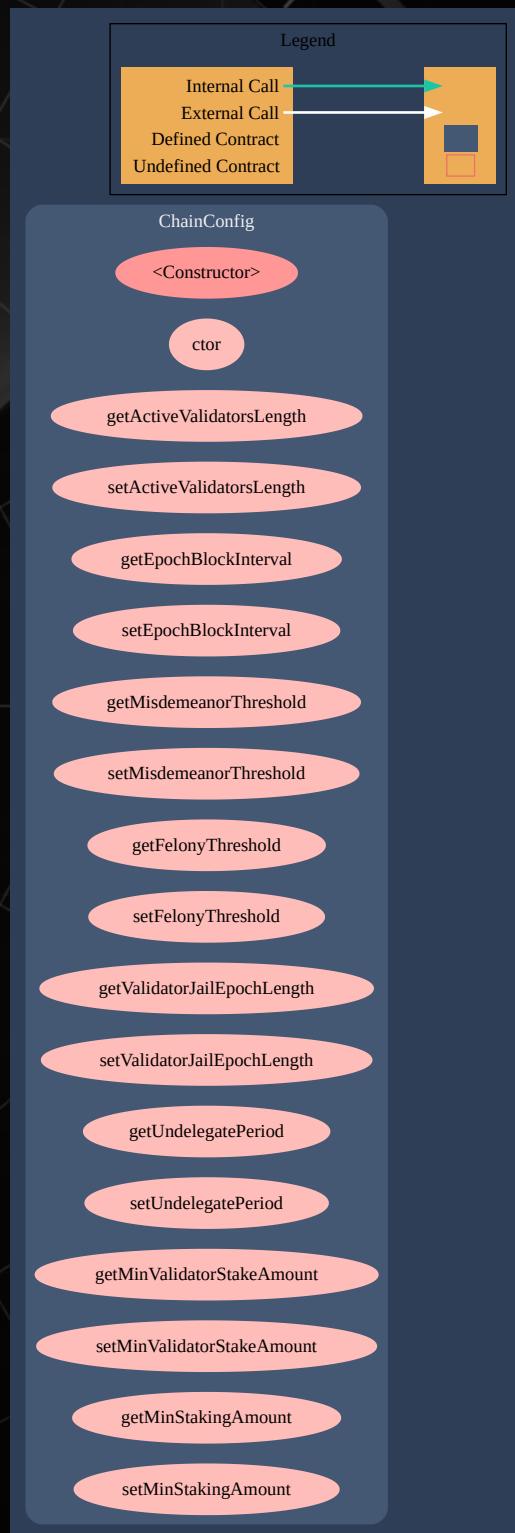
Description: A call graph of a smart contract provides a visual representation of the function calls and dependencies within the contract. It illustrates the flow of execution and the relationships between functions.

The call graph displays nodes representing individual functions and edges representing the calls made between them. The call graph allows for a comprehensive view of the contract's function hierarchy, enabling the identification of critical functions, entry points, and external dependencies.

It highlights the paths of execution, including any loops or recursive calls, which can be crucial for understanding the contract's behavior and potential risks.

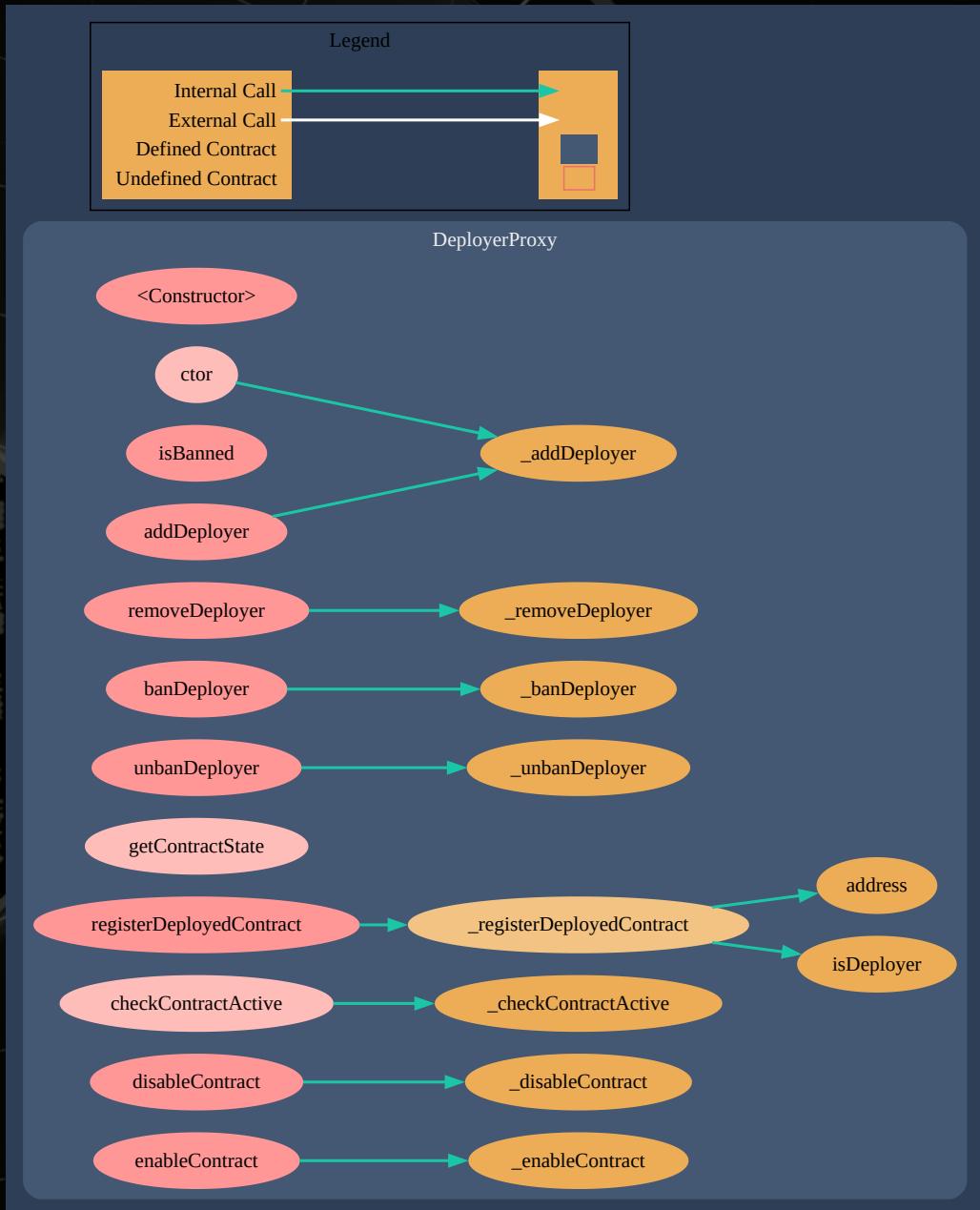
Call Graph

ChainConfig.sol



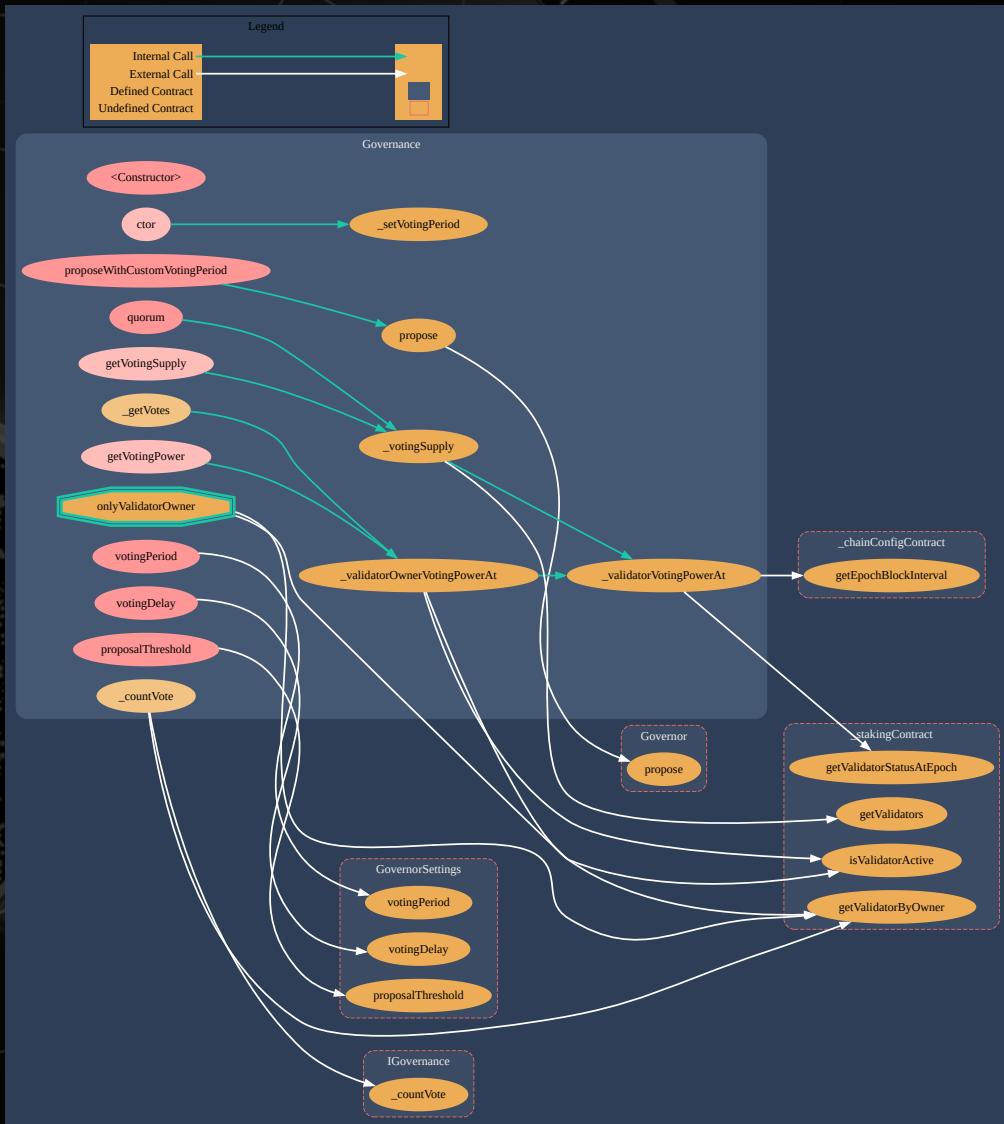
Call Graph

DeployerProxy.sol



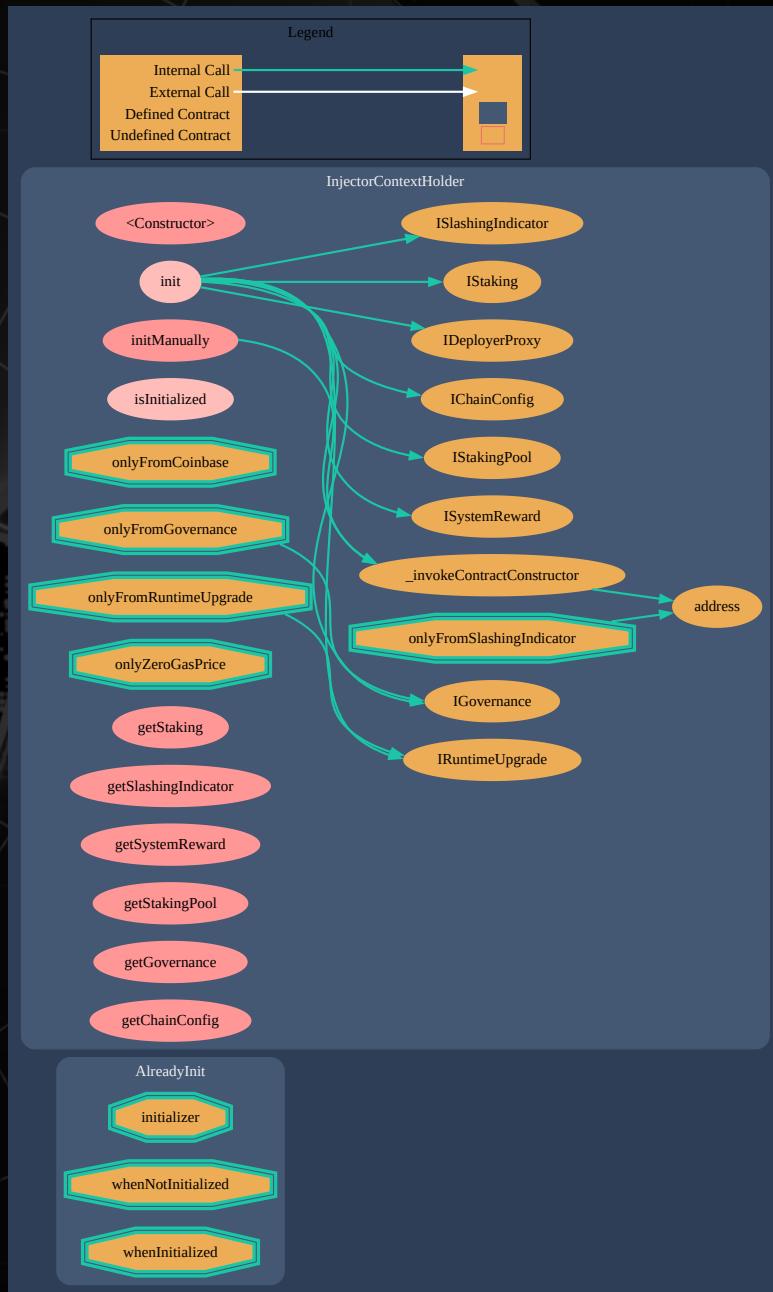
Call Graph

Governance.sol



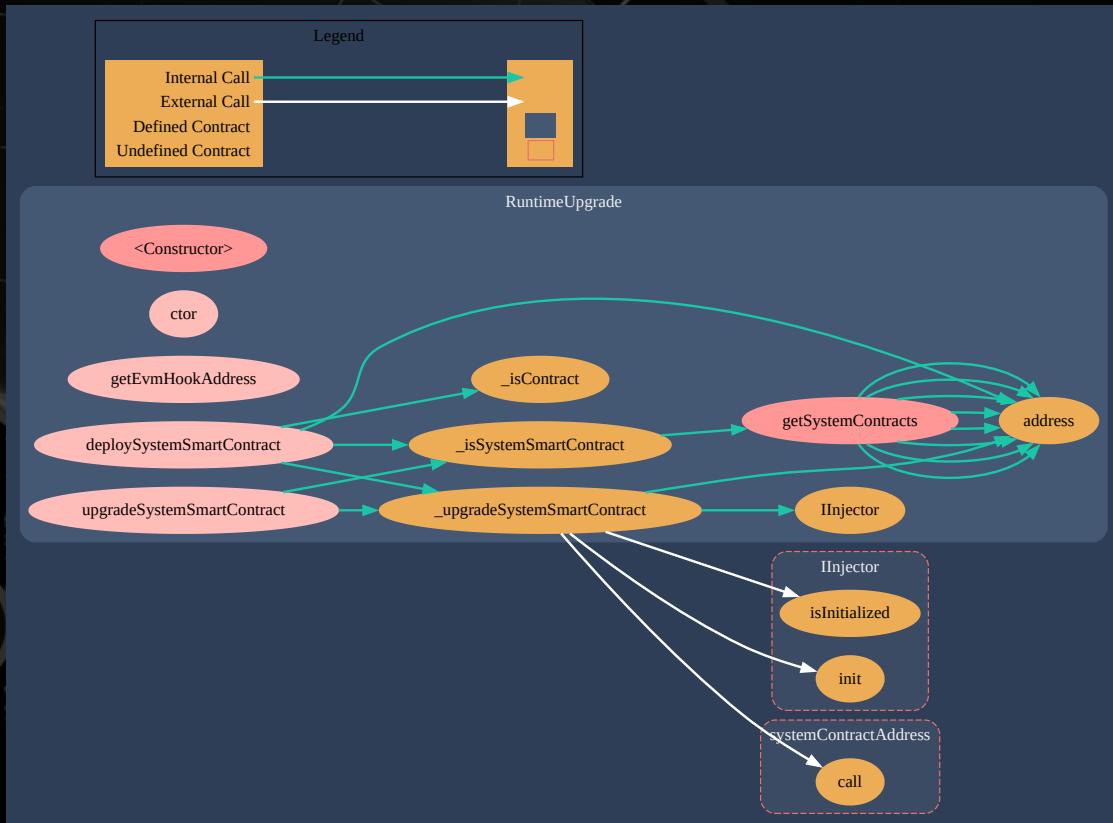
Call Graph

Injector.sol



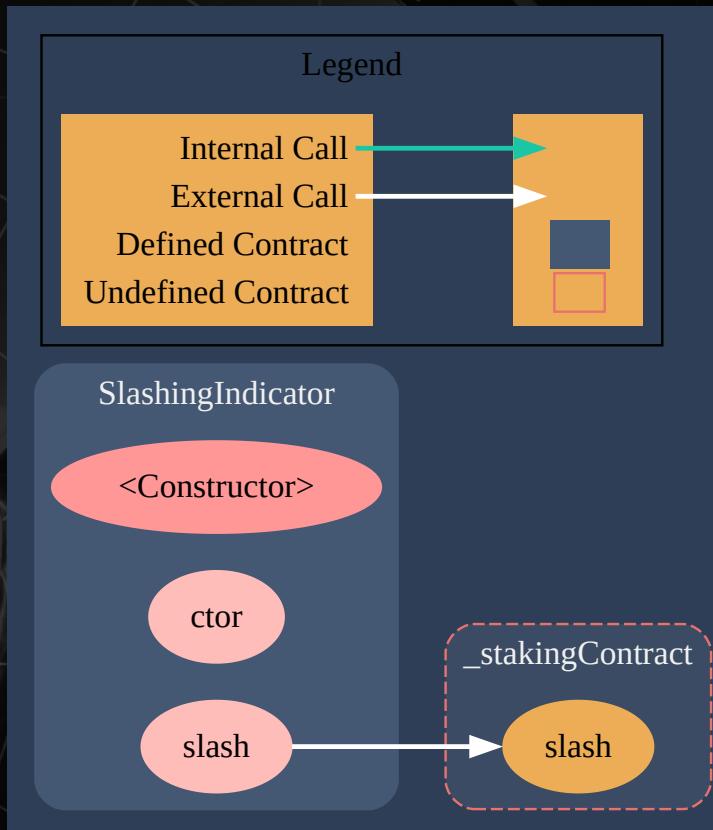
Call Graph

RuntimeUpgrade.sol



Call Graph

SlashingIndicator.sol



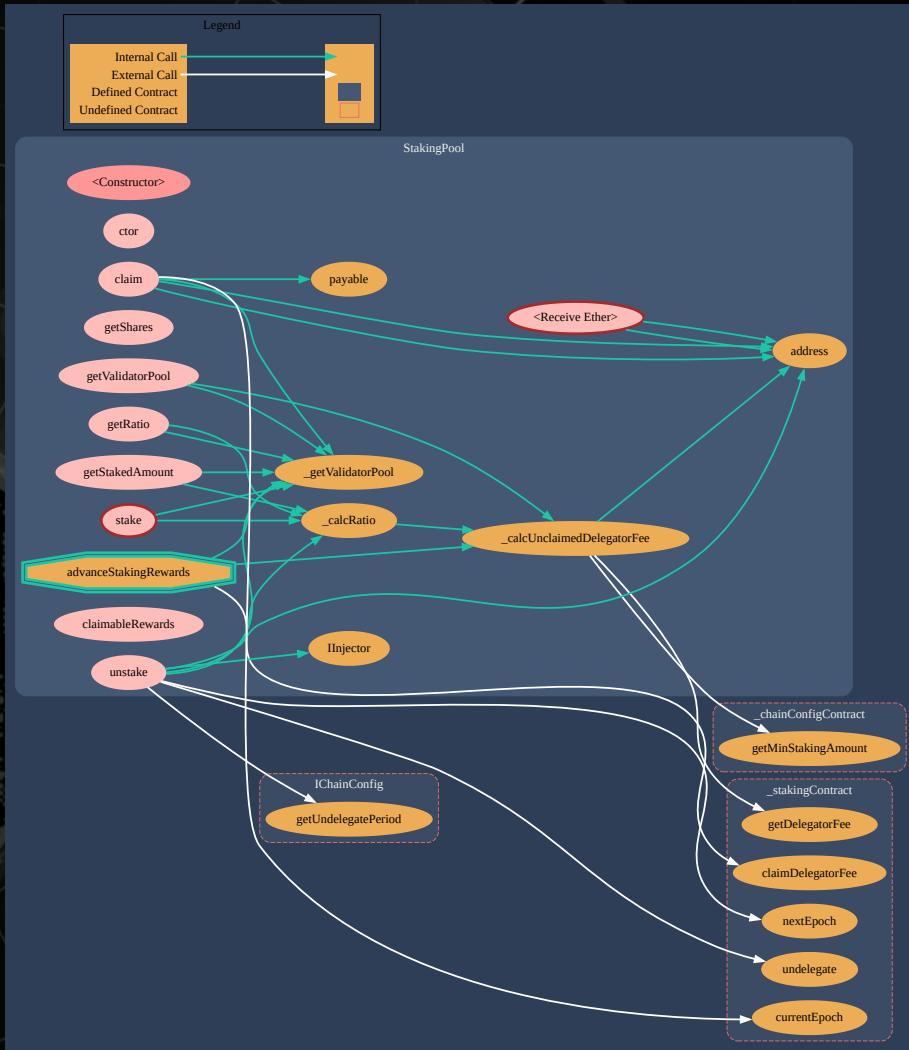
Call Graph

Staking.sol



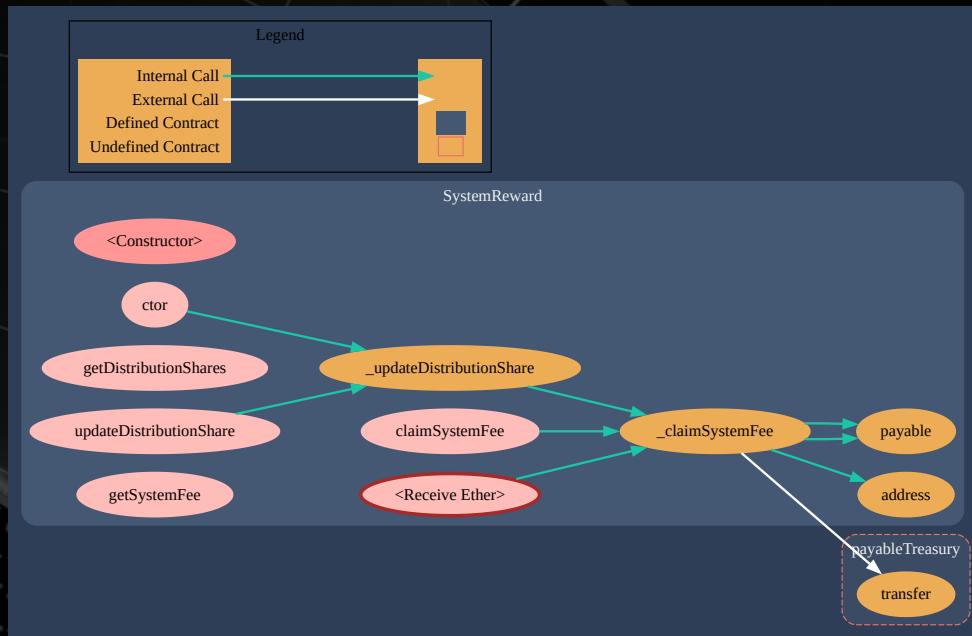
Call Graph

StakingPool.sol



Call Graph

SystemReward.sol



Contract Interaction

Issue: Contract Interaction

Severity: -

Location: General

Description: A contract interaction graph provides a visual representation of the relationships and interactions between different smart contracts within an ecosystem. It shows how contracts interact with each other through function calls, events, and state variables.

Readers can visualize the relationships and dependencies between contracts, ensuring a comprehensive analysis of the smart contract ecosystem.

The graph can be used to highlight potential security risks, communication challenges, or optimization opportunities arising from the contract interactions.

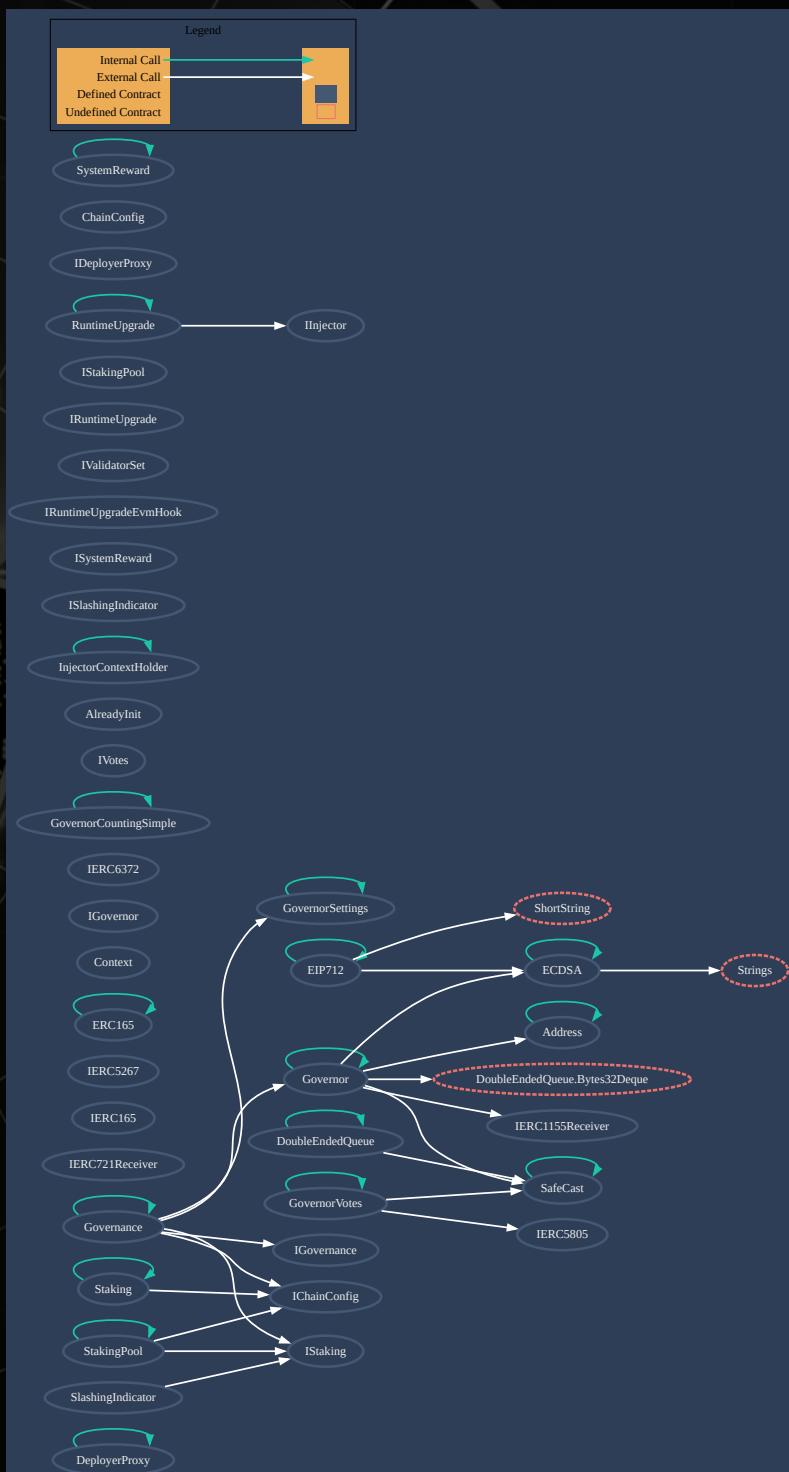
Contract Interaction

ChainConfig.sol



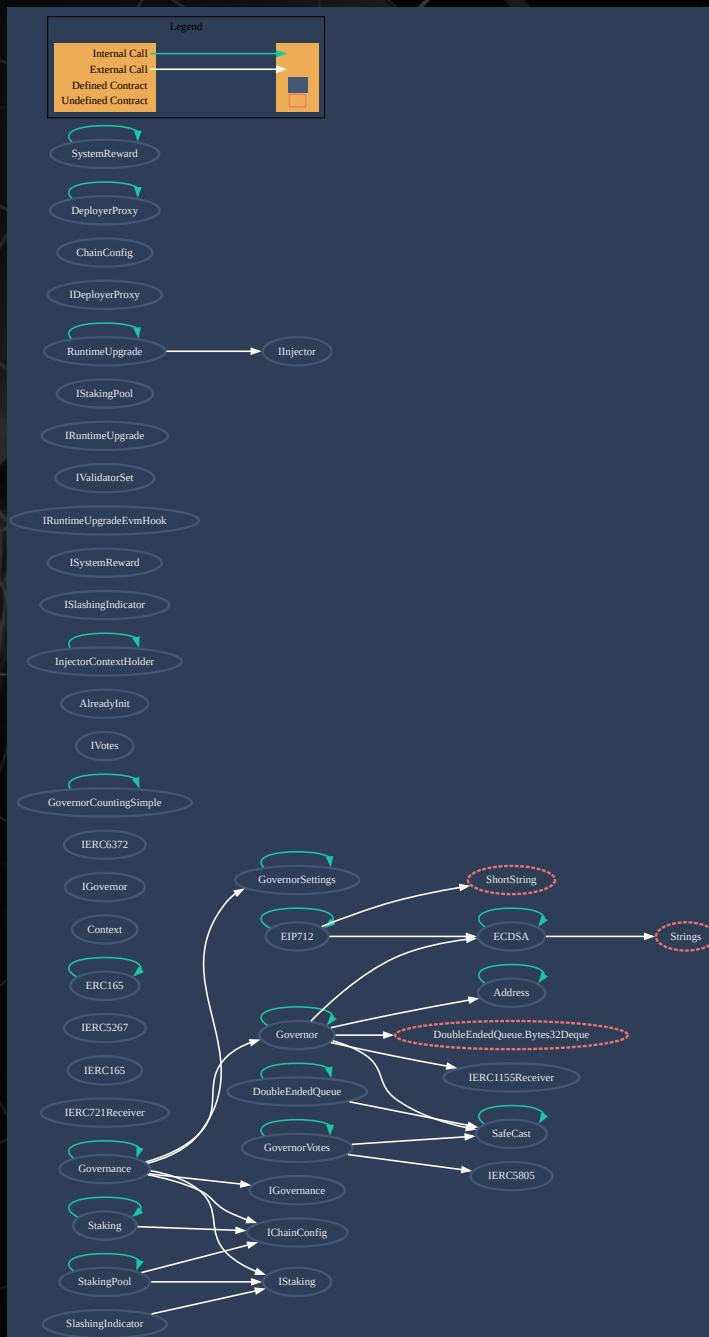
Contract Interaction

DeployerProxy.sol



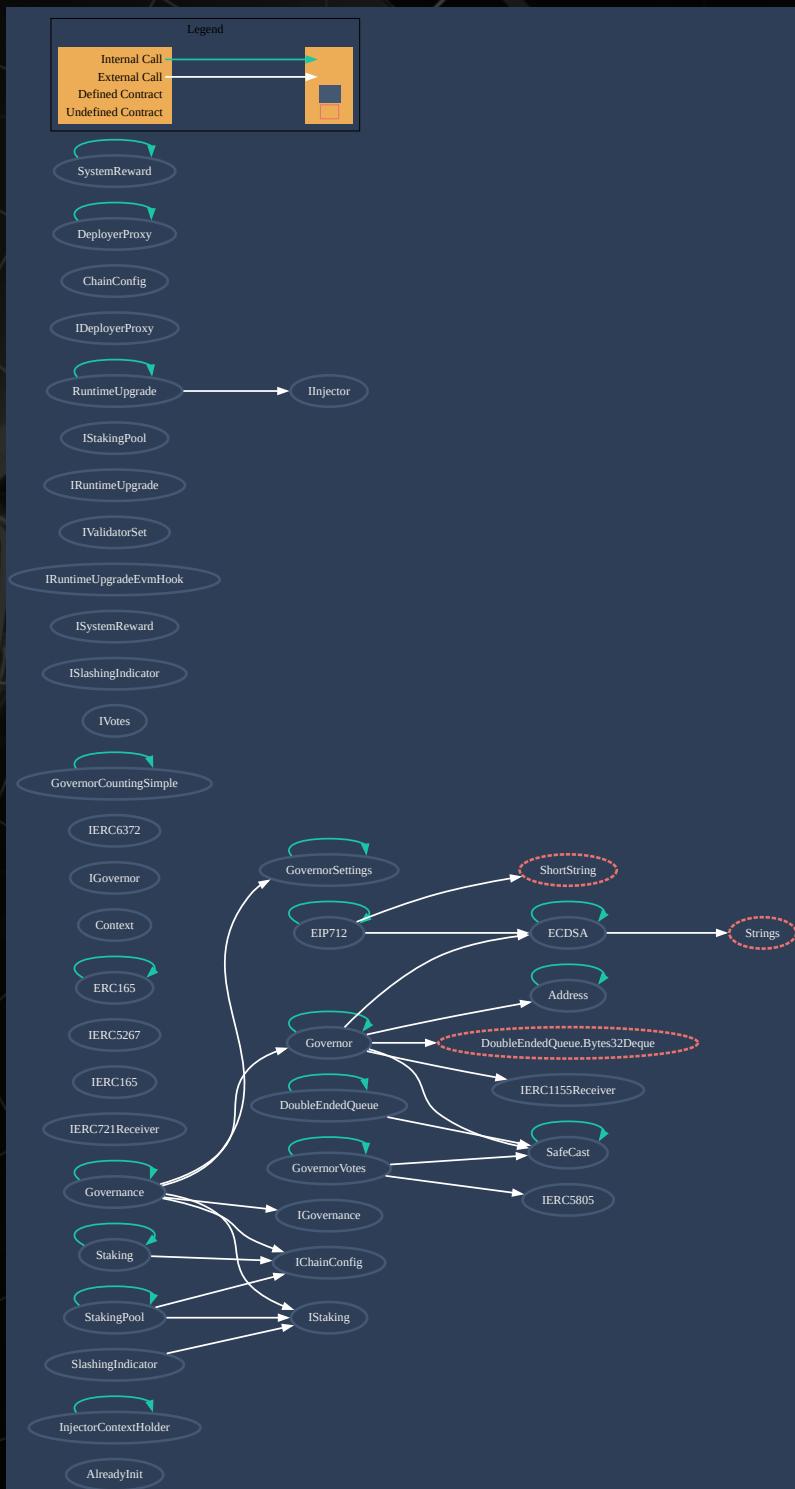
Contract Interaction

Governance.sol



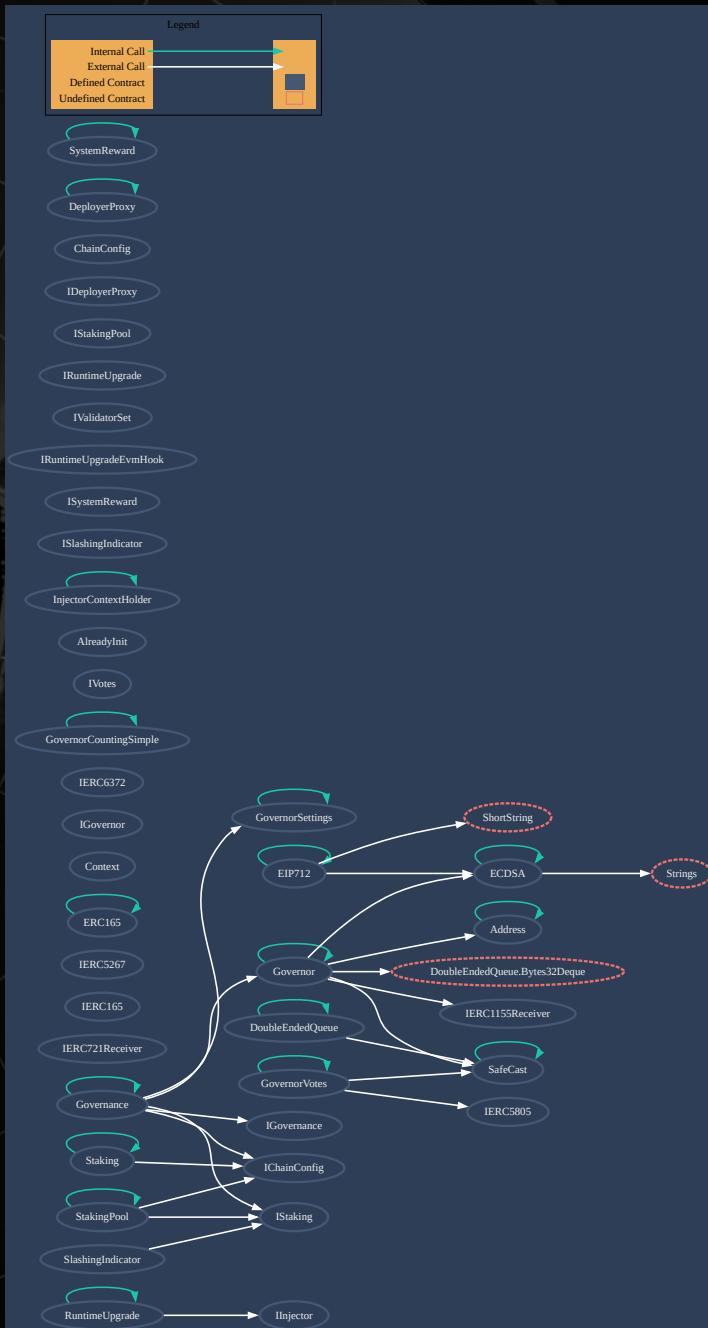
Contract Interaction

Injector.sol



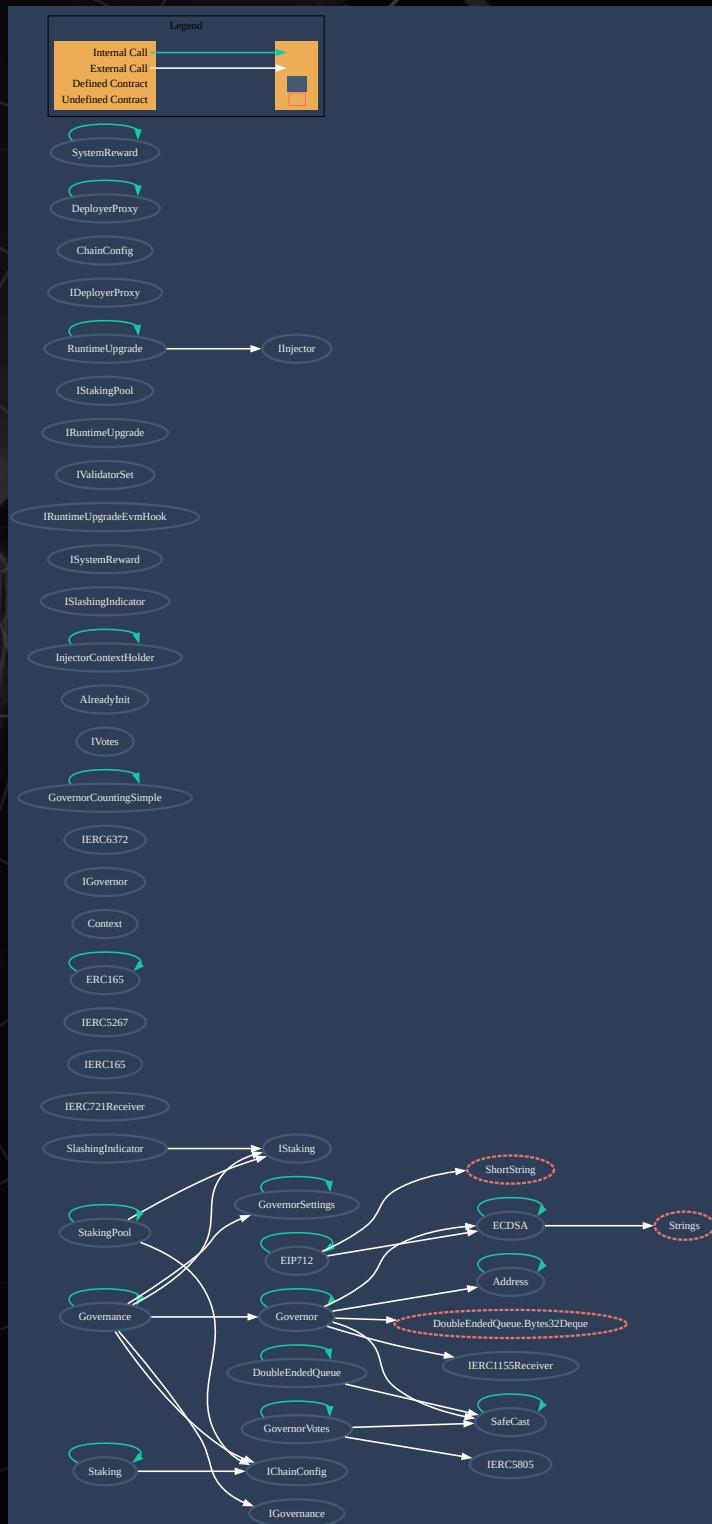
Contract Interaction

RuntimeUpgrade.sol



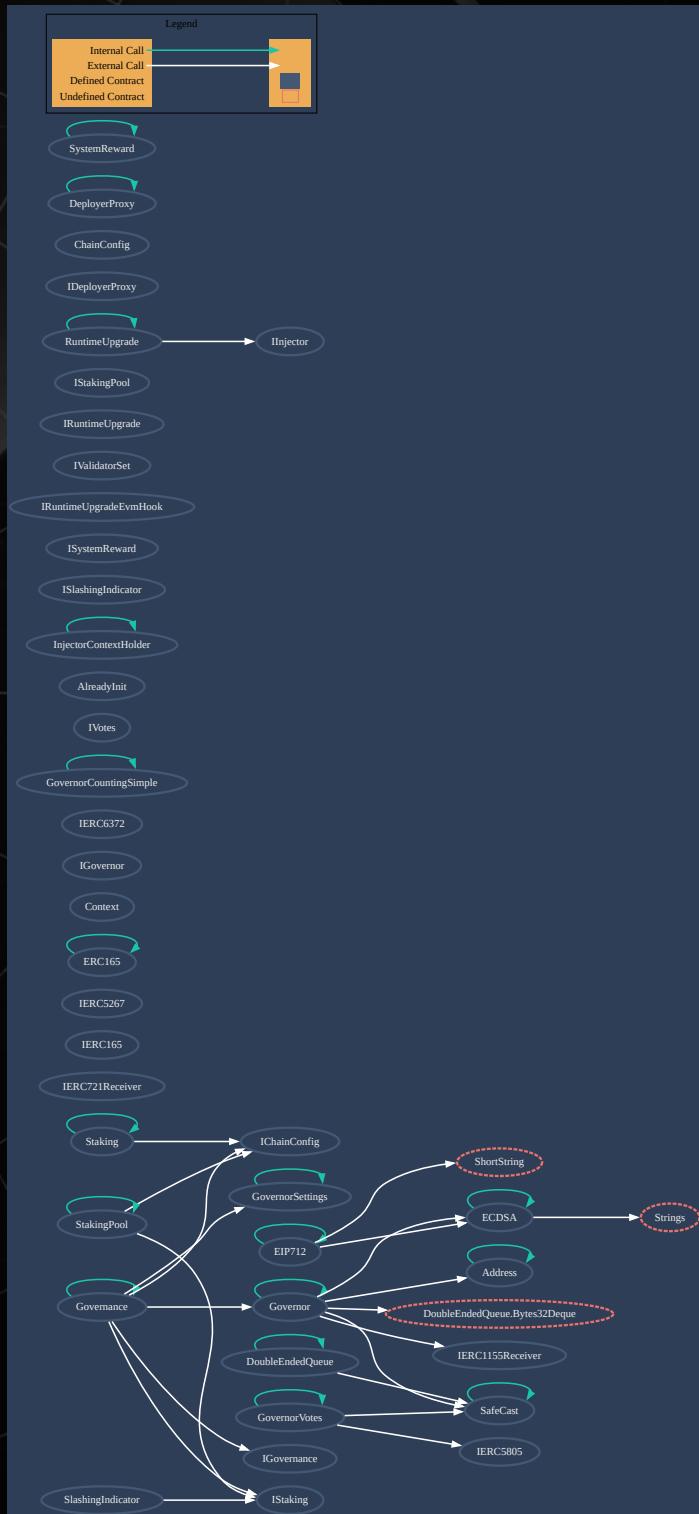
Contract Interaction

SlashingIndicator.sol



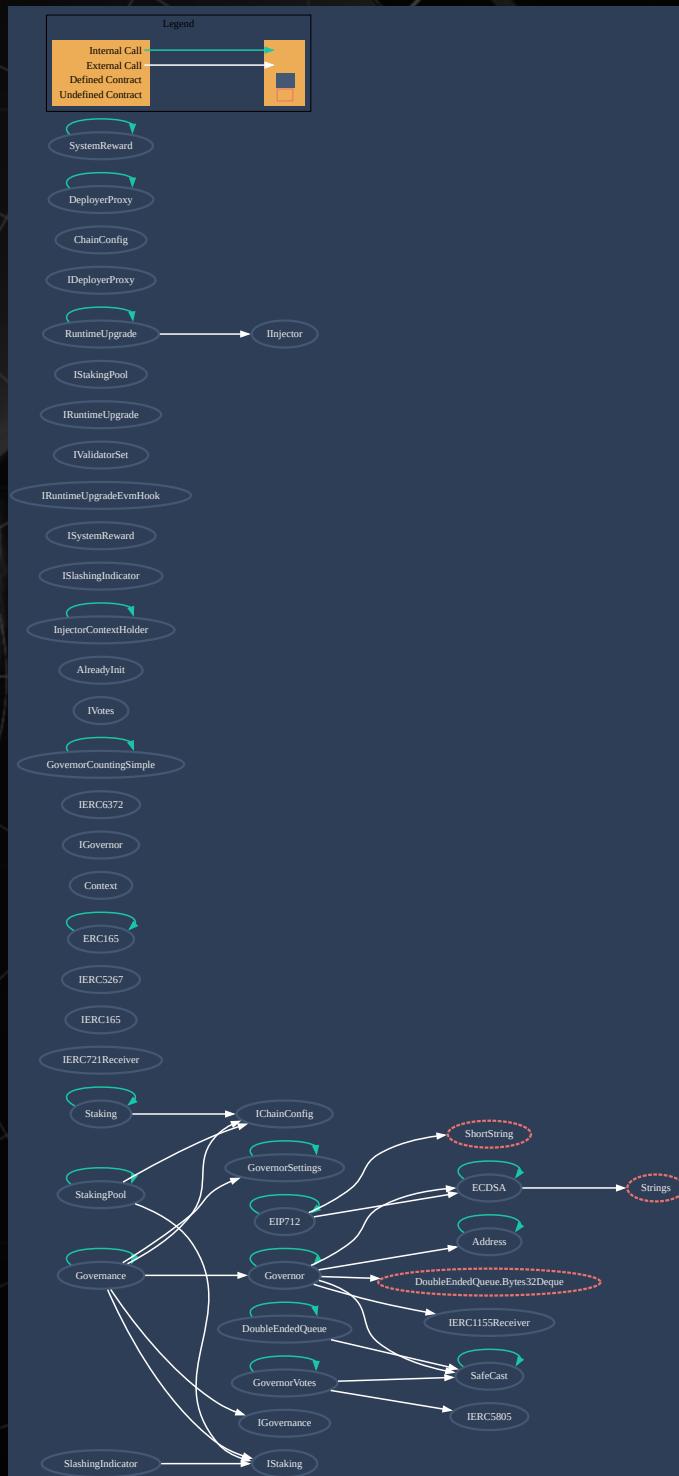
Contract Interaction

Staking.sol



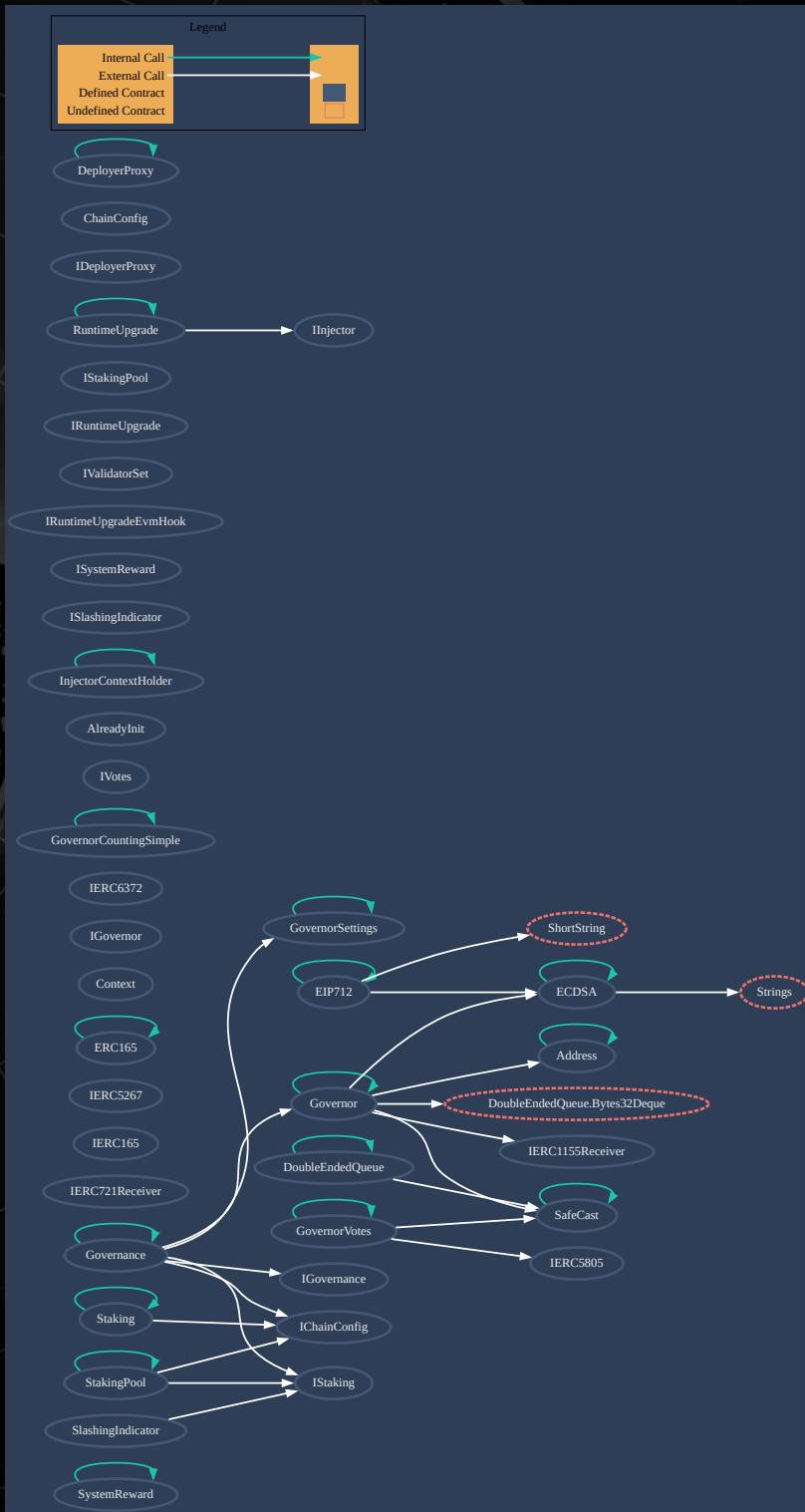
Contract Interaction

StakingPool.sol



Contract Interaction

SystemReward.sol



Analysis

Issue: Cyclomatic Complexity

Severity: Lowest / Code Style / Optimization

Location: General, Staking.sol (L320-385)

Description: In the scope of this audit, after analyzing the cyclomatic complexity of the functions present in the contract, we can see that the majority of the functions have a complexity of 1 to 3.

This indicates that the functions in the contract are relatively simple and easy to understand. A cyclomatic complexity of 1 to 3 suggests a limited number of decision points and loops, which helps reduce the overall complexity of the contract. This facilitates contract maintenance and decreases the risk of errors related to excessive complexity.

However, in the Staking.sol contract, we detected that the **_claimDelegatorRewardsAndPendingUndelegates** function has a high cyclomatic complexity (13). We recommend splitting the function into several smaller subroutines if possible.

It is important to note that cyclomatic complexity alone does not guarantee absolute security of the contract.

Analysis

Issue: Solidity Naming Convention

Severity: Lowest / Code Style / Optimization

Location: General

Description: Solidity defines a naming convention that should be followed. When widely embraced and utilized, naming conventions may be quite effective. Different conventions can be used to express crucial meta information that would otherwise be unavailable.

Comment: The naming standard defined by Solidity should be followed. In a Solidity file, the following naming rules should be followed:

Contracts should be in CapWords.

Functions and parameters should be in mixedCase.

Constants should be in UPPER_CASE_WITH_UNDERSCORES

Analysis

Issue: Usage of Inline Assembly

Severity: Lowest / Code Style / Optimization

Location: General

Description: Inline assembly is a way to access the Ethereum Virtual Machine at a low level. This discards several important safety features in Solidity and could incur in some risks should they be used incorrectly.

Comment: Using inline assembly in Solidity is discouraged as it grants direct access to the Ethereum Virtual Machine (EVM) at a low level, which could bypass important safety features of the Solidity language. It is best to rely on higher-level Solidity constructs and established libraries to ensure safer and more secure contract development.

Analysis

Issue: Contract Size

Severity: Lowest / Code Style / Optimized Practice

Location: Staking.sol

Description: Contract code size exceeds 24576 bytes (a limit introduced in Spurious Dragon).

Comment: If needed, consider enabling the optimizer (with a low "runs" value!), turning off revert strings, removing abundant code or using libraries.

Testing Standards

The goal of the audit was to find any potential smart contract security problems and vulnerabilities.

The information in this report should be used to understand the smart contract's risk exposure and as a guide to improving the smart contract's security posture by addressing the concerns that were discovered.

The blockchain platform is used to deploy and execute smart contracts. The platform, its programming language, and other smart contract-related applications all have vulnerabilities that may be exploited. As a result, the audit cannot ensure the audited smart contract(s) explicit security. Audits can't make statements or warranties on security of the code. It also cannot be deemed an adequate assessment of the code's utility and safety, bug-free status, or any statements of the smart contract. While we did our best in completing the study and publishing this report, it is crucial to emphasize that you should not rely only on it; we advocate all projects doing many independent audits and participating in a public bug bounty program to assure smart contract security.

Testing Standards

1. Gather all relevant data.
2. Perform a preliminary visual examination of all documents and contracts.
3. Find security holes with specialist tools & manual review with independent experts.
4. Create and distribute a report.



SAULIDITY



Smart Contract
Audit



saulidity.com



Saulidity



@Saulidity