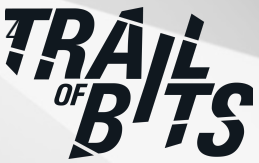

Frax Finance Governance Protocol

Security Assessment

July 7, 2023

Prepared for:
Sam Kazemian
Frax Finance

Prepared by: **Robert Schneider, Michael Colburn, and Kurt Willis**

About Trail of Bits

Founded in 2012 and headquartered in New York, Trail of Bits provides technical security assessment and advisory services to some of the world's most targeted organizations. We combine high-end security research with a real-world attacker mentality to reduce risk and fortify code. With 100+ employees around the globe, we've helped secure critical software elements that support billions of end users, including Kubernetes and the Linux kernel.

We maintain an exhaustive list of publications at https://github.com/trailofbits/publications, with links to papers, presentations, public audit reports, and podcast appearances.

In recent years, Trail of Bits consultants have showcased cutting-edge research through presentations at CanSecWest, HCSS, Devcon, Empire Hacking, GrrCon, LangSec, NorthSec, the O'Reilly Security Conference, PyCon, REcon, Security BSides, and SummerCon.

We specialize in software testing and code review projects, supporting client organizations in the technology, defense, and finance industries, as well as government entities. Notable clients include HashiCorp, Google, Microsoft, Western Digital, and Zoom.

Trail of Bits also operates a center of excellence with regard to blockchain security. Notable projects include audits of Algorand, Bitcoin SV, Chainlink, Compound, Ethereum 2.0, MakerDAO, Matic, Uniswap, Web3, and Zcash.

To keep up to date with our latest news and announcements, please follow @trailofbits on Twitter and explore our public repositories at https://github.com/trailofbits. To engage us directly, visit our "Contact" page at https://www.trailofbits.com/contact, or email us at info@trailofbits.com.

Trail of Bits, Inc.
228 Park Ave S #80688
New York, NY 10003
https://www.trailofbits.com
info@trailofbits.com

Notices and Remarks

Copyright and Distribution

Test Coverage Disclaimer

All activities undertaken by Trail of Bits in association with this project were performed in accordance with a statement of work and agreed upon project plan.

Security assessment projects are time-boxed and often reliant on information that may be provided by a client, its affiliates, or its partners. As a result, the findings documented in this report should not be considered a comprehensive list of security issues, flaws, or defects in the target system or codebase.

Trail of Bits uses automated testing techniques to rapidly test the controls and security properties of software. These techniques augment our manual security review work, but each has its limitations: for example, a tool may not generate a random edge case that violates a property or may not fully complete its analysis during the allotted time. Their use is also limited by the time and resource constraints of a project.

Table of Contents

Executive Summary

Engagement Overview

Frax Finance engaged Trail of Bits to review the security of its governance protocol. It is a decentralized governance protocol that enables users to create, vote on, and execute proposals related to itself and the wider Frax Finance system.

A team of three consultants conducted the review from May 15 to May 26, 2023, for a total of four engineer-weeks of effort. Our testing efforts focused on evaluating the governance process to ensure that it aligns with the Frax Finance team's intended functionality and security goals. With full access to the source code and documentation, we performed static and dynamic testing of the `frax-governance` repository, using automated and manual processes.

Observations and Impact

The Frax Finance governance system codebase is notably intricate for a governance system. Its dual pathways for proposals, voting delegation, and time-decaying voting power could introduce various unexpected interactions. While the Frax Finance team has incorporated safeguards against such interactions throughout the codebase, we were still able to pinpoint two high-severity issues within the `FraxGovernorOmega` contract's public proposal flow that could cause certain validations to be overlooked (TOB-FRAXGOV-1, TOB-FRAXGOV-5). However, any resulting proposals that bypass validations would still need to secure the `FraxGovernorOmega` contract's mandated 4% quorum during a public vote to pass.

Recommendations

Based on the codebase maturity evaluation and findings identified during the security review, Trail of Bits recommends that Frax Finance take the following steps:

- **Remediate the findings disclosed in this report.** These findings should be addressed as part of a direct remediation or as part of any refactor that may occur when addressing other recommendations.
- **Continue to improve dynamic test coverage.** Additional testing, especially using more complex techniques such as fuzz testing, can help uncover any latent edge cases and prevent regressions from being introduced in such a high-assurance piece of the Frax Finance ecosystem.

The following tables provide the number of findings by severity and category.

EXPOSURE ANALYSIS

Severity	Count
High	2
Medium	1
Low	0
Informational	4
Undetermined	1

CATEGORY BREAKDOWN

Category	Count
Access Controls	1
Data Validation	2
Patching	2
Timing	1
Undefined Behavior	1
Configuration	1

Project Summary

Contact Information

The following managers were associated with this project:

Dan Guido, Account Manager
dan@trailofbits.com

Anne Marie Barry, Project Manager
annemarie.barry@trailofbits.com

The following engineers were associated with this project:

Robert Schneider, Consultant
robert.schneider@trailofbits.com

Michael Colburn, Consultant
michael.colburn@trailofbits.com

Kurt Willis, Consultant
kurt.willis@trailofbits.com

Project Timeline

The significant events and milestones of the project are listed below.

Date	Event
May 11, 2023	Pre-project kickoff call
May 19, 2023	Status update meeting #1
May 26, 2023	Delivery of report draft
May 26, 2023	Report readout meeting
July 7, 2023	Delivery of final report

Project Goals

The engagement was scoped to provide a security assessment of the Frax Finance governance protocol. Specifically, we sought to accomplish the following goals:

- Conduct a thorough analysis of the smart contracts to identify potential security vulnerabilities, such as reentrancy opportunities, risks of arithmetic overflow/underflow, access control issues, data validation issues, and unchecked external calls.
- Evaluate the governance process to ensure that it aligns with the intended functionality and security goals. Verify that proposals are created, voted on, and executed correctly, and that the cancellation process operates as expected.
- Review the implementation of the weighted and “fractional” voting mechanism based on the governance token's snapshot mechanism. Ensure that voting power is accurately calculated and represented based on the token holdings at snapshot time.
- Assess the quorum thresholds implemented in the system to determine whether they are set at appropriate levels. Verify that the required participation and consensus levels are reasonable and not susceptible to manipulation.
- Review the governance system’s integration with the `TimelockController` contract to assess the delay mechanism for executing proposals. Check for any potential vulnerabilities related to the time delay, including the risk of manipulation or attacks during the waiting period.
- Perform a best effort analysis of the system’s interactions with external contracts, such as the Gnosis Safe module, to ensure that the integrations are secure and that all permissions and access controls are correctly implemented.
- Review any previously identified security issues or concerns and verify that they have been adequately addressed in subsequent contract versions or updates.
- Ensure that the contracts follow established security best practices, including code readability, proper use of standardized libraries, adherence to secure coding guidelines, and compliance with relevant industry standards.
- Thoroughly review the contract documentation and source code to identify any discrepancies, inconsistencies, or security issues that might have been overlooked.

Project Targets

The engagement involved a review and testing of the following target.

frax-governance

Repository	https://github.com/FraxFinance/frax-governance
Version	`4e3189bfaae1a667f8f0e993d374a04907ef7d12`
Type	Solidity
Platform	Ethereum

Project Coverage

This section provides an overview of the analysis coverage of the review, as determined by our high-level engagement goals. Our approaches included the following:

- **`FraxGovernorAlpha`:** This contract is a decentralized governance system designed to facilitate proposal creation, voting, and execution. It implements essential governance functionalities such as proposal management, voting, and proposal execution and cancellation, and it offers configurable settings that can be updated through the voting process. The contract incorporates weighted voting based on the governance token's built-in snapshot mechanism, allowing token holders to influence decisions proportionally. In addition to traditional voting, `FraxGovernorAlpha` extends the standard features to support fractional voting on proposals, enabling more nuanced decision-making. By inheriting from the `GovernorTimelockControl` contract, `FraxGovernorAlpha` introduces an essential time delay enforced by the `TimelockController` contract before successful proposals are executed. This delay empowers users to thoroughly consider and evaluate proposed updates before they take effect. `FraxGovernorAlpha` features a high quorum threshold, ensuring a significant level of participation is required to reach consensus on proposals. As a module on the underlying Gnosis Safe, it holds full control over the associated Gnosis Safe, as well as all governance parameters on the `FraxGovernorOmega` contract and itself. Notably, only veFXS holders possess the authority to call the `propose` function, incentivizing active participation from vested token holders. While reviewing this contract, we checked that a proposal's state is accurately tracked throughout its entire lifecycle, that the integration with Gnosis Safe is sound, and that quorum requirements are properly enforced.

- **`FraxGovernorOmega`:** This contract is a decentralized governance system similar to `FraxGovernorAlpha`, acting as a signer for the underlying Gnosis Safe. With a lower quorum threshold, it facilitates routine and lower-stakes changes more efficiently. The contract supports optimistic and non-optimistic proposals, requiring multiple Frax Finance team signatures for optimistic proposals that default to a successful state unless voted against by a quorum. `FraxGovernorOmega` empowers token holders to actively participate in shaping and governing the Frax Finance ecosystem, ensuring a robust and inclusive governance process. In addition to looking for the types of issues that we also looked for in `FraxGovernorAlpha`, we checked for any ways `FraxGovernorOmega` could bypass restrictions that prevent it from modifying governance system parameters.

- **`VeFxsVotingDelegation`:** This contract allows users to use veFXS tokens to delegate their voting rights to another address. It employs a checkpoint system to

monitor and record changes in delegation and voting power over time, ensuring accurate and timely tracking. The contract offers various functions, including calculating a user's voting weight, retrieving past voting weights, and retrieving the total supply of veFXS tokens. Moreover, users can delegate their voting power using a signature, which is a convenient method that does not require a direct transaction. Overall, this contract provides a flexible framework for users to delegate their voting power via veFXS tokens while ensuring that delegation and voting power are accurately tracked. During our review of this contract, we looked for any issues in the checkpointing logic that could cause voting power to be reported incorrectly, and we checked that delegation is carried out properly according to the specification.

Coverage Limitations

Because of the time-boxed nature of testing work, it is common to encounter coverage limitations. The following list outlines the coverage limitations of the engagement and indicates system elements that may warrant further review:

- The Gnosis Safe integration and management of multisig keys were not reviewed.
- The veFXS contract was not in scope for this review, but we referenced it when reviewing the voting power and delegation logic.

Codebase Maturity Evaluation

Trail of Bits uses a traffic-light protocol to provide each client with a clear understanding of the areas in which its codebase is mature, immature, or underdeveloped. Deficiencies identified here often stem from root causes within the software development life cycle that should be addressed through standardization measures (e.g., the use of common libraries, functions, or frameworks) or training and awareness programs.

Category	Summary	Result
Arithmetic	The system contains minimal and justified unchecked arithmetic blocks. Fuzz testing is used on most arithmetic-related code, and written explanations in the code indicate that unchecked blocks will not overflow or underflow. To increase confidence in arithmetic operations, we recommend significantly increasing the number of fuzz testing runs to allow potential bugs with more complex scenarios to be surfaced. Add public-facing documentation that more specifically defines why arithmetic bounds and ranges are safe in unchecked blocks. Also, add public-facing documentation explaining why changes to certain data types from their original implementations, such as the following, were made: • The unchecked block in the `_calculateCheckpoint` function of `VeFxsVotingDelegation.sol` • The data type of the `ProposalCore.voteStart` variable in `Governor.sol`	**Moderate**
Auditing	We observed that the system emits events for all critical functions, contributing to system transparency, traceability, and monitoring. However, to further enhance system tracking and auditing, an off-chain monitoring system should be developed. Moreover, the system lacks documentation to guide users on implementing a monitoring system themselves or on reviewing logs to audit potential system failures. Finally, an incident response plan was not provided; such a plan should be created to effectively manage system failures and security breaches.	**Moderate**

Authentication / Access Controls	Access controls are applied to privileged functions and adhere to the principle of least privilege for system components. Clear documentation on the various actors and their respective privileges was provided. The system employs a multisig wallet that ensures a lost key from a single signer does not compromise the system. Tests for access control protections are included, but they could be further expanded to cover more corner cases and known attack vectors to increase the contracts' resilience to potential threats (TOB-FRAXGOV-5). Additionally, more documentation can be added to the code that explains roles and privileges on the functions where they are applied.	**Satisfactory**
Complexity Management	The system consists of two governance contracts that serve different purposes within the wider system. Both contracts inherit from a complex inheritance tree, contain functions with the same name, and have functions with similar scopes. Both contracts interact with a Gnosis Safe, which introduces further complexity into the system. We found a high-severity issue regarding the management of an allowlist for those Gnosis Safe interactions (TOB-FRAXGOV-1). We found another high-severity issue involving an important check in the `propose` function of the `FraxGovernorOmega` contract that is undermined by a function in the `Governor` contract near the bottom of the inheritance chain (TOB-FRAXGOV-5). The Frax Finance developers have made solid efforts to explain the differences between similar inherited functionality and inheritance decisions through code comments and documentation. However, many functions, state variables, and data types that have been overridden or altered from their initial implementations are not always fully explained or explained at all. The code would benefit from a flatter architecture or more inline code comments and public-facing documentation on inheritance and design decisions. In particular, the inline code comments should distinguish between the two governance contracts themselves, all the way down the inheritance chain, especially for functions with overlapping names or responsibilities.	**Weak**

Decentralization	Risks related to trusted parties are documented. Users have a clear path to exit the system before changes take effect. Privileged actors cannot unilaterally move or trap funds within the protocol. Frax Finance provided a list of system actors and their privileges; however, we recommend adding inline code comments or public-facing documentation regarding access controls. We also recommend providing more options or alerts for users when critical configuration parameters are updated to ensure user awareness and control.	**Satisfactory**
Documentation	The provided documentation is clear and unambiguous, the system architecture is well documented through diagrams and similar constructs, and critical system components are identified. Furthermore, Frax Finance provided documentation on some risks and system limitations, reflecting a level of risk awareness. However, more public-facing documentation and inline code comments for end users and developers are needed. Our recommendations for additions to the documentation in the Arithmetic, Access Controls, Complexity Management, and Auditing sections of this table would enhance the system's usability and users' and developers' understanding of the system.	**Moderate**
Front-Running Resistance	The system is resistant to transaction ordering. Permissionless functions exist, but the user's influence over such functions is limited.	**Satisfactory**
Low-Level Manipulation	Low-level assembly blocks are used in two locations, but they are kept to a minimum and used in a straightforward way to unpack voting data and to validate the format of a signature.	**Satisfactory**
Testing and Verification	The most critical functions have positive and negative test paths covered, and all provided tests for the codebase pass. Code coverage is applied to unit tests, and the reports are retrievable. Fuzz testing is employed for important components, and testing is integrated into a CI/CD pipeline. However, we recommend expanding the number of fuzz testing runs to more effectively detect complex bugs and edge cases. Expanding the fuzz tests beyond the	**Moderate**

	`VeFxsVotingDelegation` contract to include elements of the governance contracts would also be beneficial. Also, a document describing all system invariant rules, and their subsequent implementation in the fuzz testing suite, would greatly enhance clarity and confidence in the system.	

Summary of Findings

The table below summarizes the findings of the review, including type and severity details.

ID	Title	Type	Severity
1	Race condition in FraxGovernorOmega target validation	Timing	High
2	Vulnerable project dependency	Patching	Undetermined
3	Replay protection missing in castVoteWithReasonAndParamsBySig	Data Validation	Medium
4	Ability to lock any user's tokens using deposit_for	Data Validation	Informational
5	The relay function can be used to call critical safe functions	Access Controls	High
6	Votes can be delegated to contracts	Undefined Behavior	Informational
7	Lack of public documentation regarding voting power expiry	Patching	Informational
8	Spamming risk in propose functions	Configuration	Informational

Detailed Findings

1. Race condition in FraxGovernorOmega target validation

Severity: **High**	Difficulty: **High**
Type: Timing	Finding ID: TOB-FRAXGOV-1
Target: `src/FraxGovernorOmega.sol`	

Description

The `FraxGovernorOmega` contract is intended for carrying out day-to-day operations and less sensitive proposals that do not adjust system governance parameters. Proposals directly affecting system governance are managed in the `FraxGovernorAlpha` contract, which has a much higher quorum requirement (40%, compared with `FraxGovernorOmega`'s 4% quorum requirement). When a new proposal is submitted to the `FraxGovernorOmega` contract through the `propose` or `addTransaction` function, the target address of the proposal is checked to prevent proposals from interacting with sensitive functions in allowlisted safes outside of the higher quorum flow (figure 1.1). However, if a proposal to allowlist a new safe is pending in `FraxGovernorAlpha`, and another proposal that interacts with the pending safe is preemptively submitted through `FraxGovernorOmega.propose`, the proposal would pass this check, as the new safe would not yet have been added to the allowlist.

```
/// @notice The ```propose``` function is similar to OpenZeppelin's ```propose()```
with minor changes
/// @dev Changes include: Forbidding targets that are allowlisted Gnosis Safes
/// @return proposalId Proposal ID
function propose(
    address[] memory targets,
    uint256[] memory values,
    bytes[] memory calldatas,
    string memory description
) public override returns (uint256 proposalId) {
    _requireSenderAboveProposalThreshold();

    for (uint256 i = 0; i < targets.length; ++i) {
        address target = targets[i];
        // Disallow allowlisted safes because Omega would be able to call
safe.approveHash() outside of the
        // addTransaction() / execute() / rejectTransaction() flow
        if ($safeRequiredSignatures[target] != 0) {
            revert IFraxGovernorOmega.DisallowedTarget(target);
```

```
        }
    }
```

Figure 1.1: The target validation logic in the `FraxGovernorOmega` contract's `propose` function

This issue provides a short window of time in which a proposal to update governance parameters that is submitted through `FraxGovernorOmega` could pass with the contract's 4% quorum, rather than needing to go through `FraxGovernorAlpha` and its 40% quorum, as intended. Such an exploit would also require cooperation from the safe owners to execute the approved transaction. As the vast majority of operations in the `FraxGovernorOmega` process will be optimistic proposals, the community may not monitor the contract as comprehensively as `FraxGovernorAlpha`, and a minority group of coordinated veFXS holders could take advantage of this loophole.

Exploit Scenario

A `FraxGovernorAlpha` proposal to add a new Gnosis Safe to the allowlist is being voted on. In anticipation of the proposal's approval, the new safe owner prepares and signs a transaction on this new safe for a contentious or previously vetoed action. Alice, a veFXS holder, uses `FraxGovernorOmega.propose` to initiate a proposal to approve the hash of this transaction in the new safe. Alice coordinates with enough other interested veFXS holders to reach the required quorum on the proposal. The proposal passes, and the new safe owner is able to update governance parameters without the consensus of the community.

Recommendations

Short term, add additional validation to the end of the proposal lifecycle to detect whether the target has become an allowlisted safe.

Long term, when designing new functionality, consider how this type of time-of-check to time-of-use mismatch could affect the system.

2. Vulnerable project dependency	
Severity: **Undetermined**	Difficulty: **High**
Type: Patching	Finding ID: TOB-FRAXGOV-2
Target: `package.json`	

Description

Although dependency scans did not uncover a direct threat to the project codebase, `npm audit` identified a dependency with a known vulnerability, the `yaml` library. Due to the sensitivity of the deployment code and its environment, it is important to ensure that dependencies are not malicious. Problems with dependencies in the JavaScript community could have a significant effect on the project system as a whole. The output detailing the identified issue is provided below:

Dependency	Version	ID	Description
`yaml`	>=2.0.0-5, <2.2.2	CVE-2023-2251	Uncaught exception in GitHub repository `eemeli/yaml`

Table 2.1: The issue identified by running `npm audit`

Exploit Scenario

Alice installs the dependencies for the Frax Finance governance protocol, including the vulnerable `yaml` library, on a clean machine. She subsequently uses the library, which fails to throw an error while formatting a `yaml` configuration file, impacting important data that the system needs to run as intended.

Recommendations

Short term, ensure that dependencies are up to date. Several node modules have been documented as malicious because they execute malicious code when installing dependencies to projects. Keep modules current and verify their integrity after installation.

Long term, consider integrating automated dependency auditing into the development workflow. If dependencies cannot be updated when a vulnerability is disclosed, ensure that the project codebase does not use and is not affected by the vulnerable functionality of the dependency.

3. Replay protection missing in castVoteWithReasonAndParamsBySig

Severity: **Medium**	Difficulty: **Medium**
Type: Data Validation	Finding ID: TOB-FRAXGOV-3
Target: `Governor.sol`	

Description

The `castVoteWithReasonAndParamsBySig` function does not include a voter nonce, so transactions involving the function can be replayed by anyone.

Votes can be cast through signatures by encoding the vote counts in the `params` argument.

```
function castVoteWithReasonAndParamsBySig(
    uint256 proposalId,
    uint8 support,
    string calldata reason,
    bytes memory params,
    uint8 v,
    bytes32 r,
    bytes32 s
) public virtual override returns (uint256) {
    address voter = ECDSA.recover(
        _hashTypedDataV4(
            keccak256(
                abi.encode(
                    EXTENDED_BALLOT_TYPEHASH,
                    proposalId,
                    support,
                    keccak256(bytes(reason)),
                    keccak256(params)
                )
            )
        ),
        v,
        r,
        s
    );

    return _castVote(proposalId, voter, support, reason, params);
}
```

Figure 3.1: The `castVoteWithReasonAndParamsBySig` function does not include a nonce. (`Governor.sol#L508-L535`)

The `castVoteWithReasonAndParamsBySig` function calls the `_countVoteFractional` function in the `GovernorCountingFractional` contract, which keeps track of partial votes. Unlike `_countVoteNominal`, `_countVoteFractional` can be called multiple times, as long as the voter's total voting weight is not exceeded.

Exploit Scenario

Alice has 100,000 voting power. She signs a message, and a relayer calls `castVoteWithReasonAndParamsBySig` to vote for one "yes" and one "abstain". Eve sees this transaction on-chain and replays it for the remainder of Alice's voting power, casting votes that Alice did not intend to.

Recommendations

Short term, either include a voter nonce for replay protection or modify the `_countVoteFractional` function to require that `_proposalVotersWeightCast[proposalId][account]` equals `0`, which would allow votes to be cast only once.

Long term, increase the test coverage to include cases of signature replay.

4. Ability to lock any user's tokens using deposit_for

Severity: **Informational**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-FRAXGOV-4
Target: `veFXS.vy`	

Description

The `deposit_for` function can be used to lock anyone's tokens given sufficient token approvals and an existing lock.

```
@external
@nonreentrant('lock')
def deposit_for(_addr: address, _value: uint256):
    """
    @notice Deposit `_value` tokens for `_addr` and add to the lock
    @dev Anyone (even a smart contract) can deposit for someone else, but
         cannot extend their locktime and deposit for a brand new user
    @param _addr User's wallet address
    @param _value Amount to add to user's lock
    """
    _locked: LockedBalance = self.locked[_addr]

    assert _value > 0  # dev: need non-zero value
    assert _locked.amount > 0, "No existing lock found"
    assert _locked.end > block.timestamp, "Cannot add to expired lock. Withdraw"

    self._deposit_for(_addr, _value, 0, self.locked[_addr], DEPOSIT_FOR_TYPE)
```

Figure 4.1: The `deposit_for` function can be used to lock anyone's tokens. (`test/veFXS.vy#L458-L474`)

The same issue is present in the veCRV contract for the CRV token, so it may be known or intentional.

Exploit Scenario

Alice gives unlimited FXS token approval to the veFXS contract. Alice wants to lock 1 FXS for 4 years. Bob sees that Alice has 100,000 FXS and locks all of the tokens for her. Alice is no longer able to access her 100,000 FXS.

Recommendations

Short term, make users aware of the issue in the existing token contract. Only present the user with exact approval limits when locking FXS.

5. The relay function can be used to call critical safe functions

Severity: **High**	Difficulty: **Medium**
Type: Access Controls	Finding ID: TOB-FRAXGOV-5
Target: `src/Governor.sol`	

Description

The `relay` function of the `FraxGovernorOmega` contract supports arbitrary calls to arbitrary targets and can be leveraged in a proposal to call sensitive functions on the Gnosis Safe.

```
function relay(address target, uint256 value, bytes calldata data) external payable
virtual onlyGovernance {
        (bool success, bytes memory returndata) = target.call{value: value}(data);
        Address.verifyCallResult(success, returndata, "Governor: relay reverted
without message");
    }
```

Figure 5.1: The `relay` function inherited from `Governor.sol`

The `FraxGovernorOmega` contract checks proposed transactions to ensure they do not target critical functions on the Gnosis Safe contract outside of the more restrictive `FraxGovernorAlpha` flow.

```
function propose(
        address[] memory targets,
        uint256[] memory values,
        bytes[] memory calldatas,
        string memory description
    ) public override returns (uint256 proposalId) {
        _requireSenderAboveProposalThreshold();

        for (uint256 i = 0; i < targets.length; ++i) {
            address target = targets[i];
            // Disallow allowlisted safes because Omega would be able to call
safe.approveHash() outside of the
            // addTransaction() / execute() / rejectTransaction() flow
            if ($safeRequiredSignatures[target] != 0) {
                revert IFraxGovernorOmega.DisallowedTarget(target);
            }
        }

        proposalId = _propose({ targets: targets, values: values, calldatas:
calldatas, description: description });
```

```
    }
```

Figure 5.2: The `propose` function of `FraxGovernorOmega.sol`

A malicious user can hide a call to the Gnosis Safe by wrapping it in a call to the `relay` function. There are no further restrictions on the target contract argument, which means the `relay` function can be called with calldata that targets the Gnosis Safe contract.

Exploit Scenario

Alice, a veFXS holder, submits a transaction to the `propose` function. The `targets` array contains the `FraxGovernorOmega` address, and the corresponding `calldatas` array contains an encoded call to its `relay` function. The encoded call to the `relay` function has a target address of an allowlisted Gnosis Safe and an encoded call to its `approveHash` function with a payload of a malicious transaction hash. Due to the low quorum threshold on `FraxGovernorOmega` and the shorter voting period, Alice is able to push her malicious transaction through, and it is approved by the safe even though it should not have been.

Recommendations

Short term, add a check to the `relay` function that prevents it from targeting addresses of allowlisted safes.

Long term, carefully examine all cases of user-provided inputs, especially where arbitrary targets and calldata can be submitted, and expand the unit tests to account for edge cases specific to the wider system.

6. Votes can be delegated to contracts

Severity: **Informational**	Difficulty: **Low**
Type: Undefined Behavior	Finding ID: TOB-FRAXGOV-6
Target: `VeFxsVotingDelegation.sol`	

Description

Votes can be delegated to smart contracts. This behavior contrasts with the fact that FXS tokens can be locked only in whitelisted contracts. Allowing votes to be delegated to smart contracts could lead to unexpected behavior.

By default, smart contracts are unable to gain voting power; to gain voting power, they need to be explicitly whitelisted by the Frax Finance team in the veFXS contract.

```
@internal
def assert_not_contract(addr: address):
    """
    @notice Check if the call is from a whitelisted smart contract, revert if not
    @param addr Address to be checked
    """
    if addr != tx.origin:
        checker: address = self.smart_wallet_checker
        if checker != ZERO_ADDRESS:
            if SmartWalletChecker(checker).check(addr):
                return
        raise "Smart contract depositors not allowed"
```

Figure 6.1: The contract check in veFXS.vy

This is the intended design of the voting escrow contract, as allowing smart contracts to vote would enable wrapped tokens and bribes.

The `VeFxsVotingDelegation` contract enables users to delegate their voting power to other addresses, but it does not contain a check for smart contracts. This means that smart contracts can now hold voting power, and the team is unable to disallow this.

```
function _delegate(address delegator, address delegatee) internal {
    // Revert if delegating to self with address(0), should be address(delegator)
    if (delegatee == address(0)) revert
IVeFxsVotingDelegation.IncorrectSelfDelegation();

    IVeFxsVotingDelegation.Delegation memory previousDelegation =
$delegations[delegator];
```

```
    // This ensures that checkpoints take effect at the next epoch
    uint256 checkpointTimestamp = ((block.timestamp / 1 days) * 1 days) + 1 days;

    IVeFxsVotingDelegation.NormalizedVeFxsLockInfo
        memory normalizedDelegatorVeFxsLockInfo = _getNormalizedVeFxsLockInfo({
            delegator: delegator,
            checkpointTimestamp: checkpointTimestamp
        });

    _moveVotingPowerFromPreviousDelegate({
        previousDelegation: previousDelegation,
        checkpointTimestamp: checkpointTimestamp
    });

    _moveVotingPowerToNewDelegate({
        newDelegate: delegatee,
        delegatorVeFxsLockInfo: normalizedDelegatorVeFxsLockInfo,
        checkpointTimestamp: checkpointTimestamp
    });

    // ...
}
```

Figure 6.2: The `_delegate` function in `VeFxsVotingDelegation.sol`

Exploit Scenario

Eve sets up a contract that accepts delegated votes in exchange for rewards. The contract ends up owning a majority of the FXS voting power.

Recommendations

Short term, consider whether smart contracts should be allowed to hold voting power. If so, document this fact; if not, add a check to the `VeFxsVotingDelegation` contract to ensure that addresses receiving delegated voting power are not smart contracts.

Long term, when implementing new features, consider the implications of adding them to ensure that they do not lift constraints that were placed beforehand.

7. Lack of public documentation regarding voting power expiry

Severity: **Informational**	Difficulty: **Low**
Type: Patching	Finding ID: TOB-FRAXGOV-7
Target: `VeFxsVotingDelegation.sol`	

Description

The user documentation concerning the calculation of voting power is unclear.

The Frax-Governance specification sheet provided by the Frax Finance team states, "Voting power goes to 0 at veFXS lock expiration time, this is different from `veFXS.getBalance()` which will return the locked amount of FXS after the lock has expired."

This statement is in line with the code's behavior. The `_calculateVotingWeight` function in the `VeFxsVotingDelegation` contract does not return the locked veFXS balance once a lock has expired.

```
/// @notice The ```_calculateVotingWeight``` function calculates ```account```'s
voting weight. Is 0 if they ever delegated and the delegation is in effect.
/// @param voter Address of voter
/// @param timestamp A block.timestamp, typically corresponding to a proposal
snapshot
/// @return votingWeight Voting weight corresponding to ```account```'s veFXS
balance
function _calculateVotingWeight(address voter, uint256 timestamp) internal view
returns (uint256) {
    // If lock is expired they have no voting weight
    if (VE_FXS.locked(voter).end <= timestamp) return 0;

    uint256 firstDelegationTimestamp = $delegations[voter].firstDelegationTimestamp;
    // Never delegated OR this timestamp is before the first delegation by account
    if (firstDelegationTimestamp == 0 || timestamp < firstDelegationTimestamp) {
        try VE_FXS.balanceOf({ addr: voter, _t: timestamp }) returns (uint256
_balance) {
            return _balance;
        } catch {}
    }
    return 0;
}
```

Figure 7.2: The function that calculates the voting weight in `VeFxsVotingDelegation.sol`

If a voter's lock has expired or was never created, the short-circuit condition returns zero voting power. This behavior contrasts with the `veFxs.balanceOf` function, which would return the user's last locked FXS balance.

```
@external
@view
def balanceOf(addr: address, _t: uint256 = block.timestamp) -> uint256:
    """
    @notice Get the current voting power for `msg.sender`
    @dev Adheres to the ERC20 `balanceOf` interface for Aragon compatibility
    @param addr User wallet address
    @param _t Epoch time to return voting power at
    @return User voting power
    """
    _epoch: uint256 = self.user_point_epoch[addr]
    if _epoch == 0:
        return 0
    else:
        last_point: Point = self.user_point_history[addr][_epoch]
        last_point.bias -= last_point.slope * convert(_t - last_point.ts, int128)
        if last_point.bias < 0:
            last_point.bias = 0

        unweighted_supply: uint256 = convert(last_point.bias, uint256) # Original
from veCRV
        weighted_supply: uint256 = last_point.fxs_amt + (VOTE_WEIGHT_MULTIPLIER *
unweighted_supply)
        return weighted_supply
```

Figure 7.1: The `balanceOf` function in `veFXS.vy`

This divergence should be clearly documented in the code and should be reflected in Frax Finance's public-facing documentation, which does not mention the fact that an expired lock does not hold any voting power: "Each veFXS has 1 vote in governance proposals. Staking 1 FXS for the maximum time, 4 years, would generate 4 veFXS. This veFXS balance itself will slowly decay down to 1 veFXS after 4 years, [...]".

Exploit Scenario

Alice buys FXS to be able to vote on a proposal. She is not aware that she is required to create a lock (even if expired) to have any voting power at all. She is unable to vote for the proposal.

Recommendations

Short term, modify the `VeFxsVotingDelegation` contract to reflect the desired voting power curve and/or document whether this is intended behavior.

Long term, make sure to keep public-facing documentation up to date when changes are made.

8. Spamming risk in propose functions

Severity: **Informational**	Difficulty: **Low**
Type: Configuration	Finding ID: TOB-FRAXGOV-8
Target: `FraxGovernorAlpha.sol`, `FraxGovernorBravo.sol`	

Description
Anyone with enough veFXS tokens to meet the proposal threshold can submit an unbounded number of proposals to both the `FraxGovernorAlpha` and `FraxGovernorOmega` contracts.

The only requirement for submitting proposals is that the `msg.sender` address must have a balance of veFXS tokens larger than the `_proposalThreshold` value. Once that requirement is met, a user can submit as many proposals as they would like. A large volume of proposals may create difficulties for off-chain monitoring solutions and user-interface interactions.

```
function _requireSenderAboveProposalThreshold() internal view {
    if (_getVotes(msg.sender, block.timestamp - 1, "") < proposalThreshold()) {
        revert SenderVotingWeightBelowProposalThreshold();
    }
}
```

Figure 8.1: The `_requireSenderAboveProposalThreshold` function, called by the `propose` function (`FraxGovernorBase.sol#L104-L108`)

Exploit Scenario
Mallory has 100,000 voting power. She submits one million proposals with small but unique changes to the description field of each one. The system saves one million unique proposals and emits one million `ProposalCreated` events. Front-end components and off-chain monitoring systems are spammed with large quantities of data.

Recommendations
Short term, track and limit the number of proposals a user can have active at any given time.

Long term, consider cases of user interactions beyond just the intended use cases for potential malicious behavior.

A. Vulnerability Categories

The following tables describe the vulnerability categories, severity levels, and difficulty levels used in this document.

Vulnerability Categories	
Category	**Description**
Access Controls	Insufficient authorization or assessment of rights
Auditing and Logging	Insufficient auditing of actions or logging of problems
Authentication	Improper identification of users
Configuration	Misconfigured servers, devices, or software components
Cryptography	A breach of system confidentiality or integrity
Data Exposure	Exposure of sensitive information
Data Validation	Improper reliance on the structure or values of data
Denial of Service	A system failure with an availability impact
Error Reporting	Insecure or insufficient reporting of error conditions
Patching	Use of an outdated software package or library
Session Management	Improper identification of authenticated users
Testing	Insufficient test methodology or test coverage
Timing	Race conditions or other order-of-operations flaws
Undefined Behavior	Undefined behavior triggered within the system

Severity Levels	
Severity	**Description**
Informational	The issue does not pose an immediate risk but is relevant to security best practices.
Undetermined	The extent of the risk was not determined during this engagement.
Low	The risk is small or is not one the client has indicated is important.
Medium	User information is at risk; exploitation could pose reputational, legal, or moderate financial risks.
High	The flaw could affect numerous users and have serious reputational, legal, or financial implications.

Difficulty Levels	
Difficulty	**Description**
Undetermined	The difficulty of exploitation was not determined during this engagement.
Low	The flaw is well known; public tools for its exploitation exist or can be scripted.
Medium	An attacker must write an exploit or will need in-depth knowledge of the system.
High	An attacker must have privileged access to the system, may need to know complex technical details, or must discover other weaknesses to exploit this issue.

B. Code Maturity Categories

The following tables describe the code maturity categories and rating criteria used in this document.

Code Maturity Categories	
Category	**Description**
Arithmetic	The proper use of mathematical operations and semantics
Auditing	The use of event auditing and logging to support monitoring
Authentication / Access Controls	The use of robust access controls to handle identification and authorization and to ensure safe interactions with the system
Complexity Management	The presence of clear structures designed to manage system complexity, including the separation of system logic into clearly defined functions
Cryptography and Key Management	The safe use of cryptographic primitives and functions, along with the presence of robust mechanisms for key generation and distribution
Decentralization	The presence of a decentralized governance structure for mitigating insider threats and managing risks posed by contract upgrades
Documentation	The presence of comprehensive and readable codebase documentation
Front-Running Resistance	The system's resistance to front-running attacks
Low-Level Manipulation	The justified use of inline assembly and low-level calls
Testing and Verification	The presence of robust testing procedures (e.g., unit tests, integration tests, and verification methods) and sufficient test coverage

Rating Criteria	
Rating	**Description**
Strong	No issues were found, and the system exceeds industry standards.
Satisfactory	Minor issues were found, but the system is compliant with best practices.
Moderate	Some issues that may affect system safety were found.
Weak	Many issues that affect system safety were found.
Missing	A required component is missing, significantly affecting system safety.
Not Applicable	The category is not applicable to this review.
Not Considered	The category was not considered in this review.
Further Investigation Required	Further investigation is required to reach a meaningful conclusion.

C. Multisignature Wallet Best Practices

Consensus requirements for sensitive actions, such as spending funds from a wallet, are meant to mitigate the risks of the following:

- Any one person overruling the judgment of others
- Failures caused by any one person's mistake
- Failures caused by the compromise of any one person's credentials

For example, in a 2-of-3 multisignature wallet, the authority to execute a "spend" transaction would require a consensus of two individuals in possession of two of the wallet's three private keys. For this model to be useful, the following conditions are required:

1. The private keys must be stored or held separately, and access to each one must be limited to a unique individual.
2. If the keys are physically held by third-party custodians (e.g., a bank), multiple keys should not be stored with the same custodian. (Doing so would violate requirement #1.)
3. The person asked to provide the second and final signature on a transaction (i.e., the co-signer) should refer to a pre-established policy specifying the conditions for approving the transaction by signing it with his or her key.
4. The co-signer should also verify that the half-signed transaction was generated willingly by the intended holder of the first signature's key.

Requirement #3 prevents the co-signer from becoming merely a "deputy" acting on behalf of the first signer (forfeiting the decision-making responsibility to the first signer and defeating the security model). If the co-signer can refuse to approve the transaction for any reason, the due-diligence conditions for approval may be unclear. That is why a policy for validating transactions is needed. A verification policy could include the following:

- A protocol for handling a request to co-sign a transaction (e.g., a half-signed transaction will be accepted only via an approved channel)
- An allowlist of specific addresses allowed to be the payee of a transaction
- A limit on the amount of funds spent in a single transaction or in a single day

Requirement #4 mitigates the risks associated with a single stolen key. For example, say that an attacker somehow acquired the unlocked Ledger Nano S of one of the signatories. A voice call from the co-signer to the initiating signatory to confirm the transaction would reveal that the key had been stolen and that the transaction should not be co-signed. If the signatory were under an active threat of violence, he or she could use a duress code (a code word, a phrase, or another signal agreed upon in advance) to covertly alert the others that the transaction had not been initiated willingly, without alerting the attacker.

D. Code Quality Findings

This appendix lists a finding that is not associated with specific vulnerabilities.

GovernorCountingFractional

- **Update the `GovernorCountingFractional` contract to import Frax Finance's modified `Governor.sol` contract.** `GovernorCountingFractional` imports the unmodified OpenZeppelin `Governor` contract, though it does not use it directly or inherit from it. In the event of more extensive future modifications, it may become unclear to readers or tooling which `Governor` contract should be correctly referenced.

E. Incident Response Plan Recommendations

This section provides recommendations on formulating an incident response plan.

- **Identify the parties (either specific people or roles) responsible for implementing the mitigations when an issue occurs (e.g., deploying smart contracts, pausing contracts, upgrading the front end, etc.).**
- **Clearly describe the intended contract deployment process.**
- **Outline the circumstances under which Frax Finance will compensate users affected by an issue (if any).**
    - Issues that warrant compensation could include an individual or aggregate loss or a loss resulting from user error, a contract flaw, or a third-party contract flaw.
- **Document how the team plans to stay up to date on new issues that could affect the system; awareness of such issues will inform future development work and help the team secure the deployment toolchain and the external on-chain and off-chain services that the system relies on.**
    - Identify sources of vulnerability news for each language and component used in the system, and subscribe to updates from each source. Consider creating a private Discord channel in which a bot will post the latest vulnerability news; this will provide the team with a way to track all updates in one place. Lastly, consider assigning certain team members to track news about vulnerabilities in specific components of the system.
- **Determine when the team will seek assistance from external parties (e.g., auditors, affected users, other protocol developers, etc.) and how it will onboard them.**
    - Effective remediation of certain issues may require collaboration with external parties.
- **Define contract behavior that would be considered abnormal by off-chain monitoring solutions.**

It is best practice to perform periodic dry runs of scenarios outlined in the incident response plan to find omissions and opportunities for improvement and to develop “muscle memory.” Additionally, document the frequency with which the team should perform dry runs of various scenarios, and perform dry runs of more likely scenarios more

regularly. Create a template to be filled out with descriptions of any necessary improvements after each dry run.

J. Fix Review Results

When undertaking a fix review, Trail of Bits reviews the fixes implemented for issues identified in the original report. This work involves a review of specific areas of the source code and system configuration, not comprehensive analysis of the system.

On June 28, 2023, Trail of Bits reviewed the fixes and mitigations implemented by the Frax Finance team for the issues identified in this report. We reviewed each fix to determine its effectiveness in resolving the associated issue.

In summary, of the eight issues described in this report, Frax Finance has resolved four issues, has partially resolved one issue, and has not resolved the remaining three issues. For additional information, please see the Detailed Fix Review Results below.

ID	Title	Severity	Status
1	Race condition in FraxGovernorOmega target validation	High	Resolved
2	Vulnerable project dependency	Undetermined	Resolved
3	Replay protection missing in castVoteWithReasonAndParamsBySig	Medium	Resolved
4	Ability to lock any user's tokens using deposit_for	Informational	Unresolved
5	The relay function can be used to call critical safe functions	High	Resolved
6	Votes can be delegated to contracts	Informational	Unresolved
7	Lack of public documentation regarding voting power expiry	Informational	Unresolved
8	Spamming risk in propose functions	Informational	Partially Resolved

Detailed Fix Review Results

TOB-FRAXGOV-1: Race condition in FraxGovernorOmega target validation
Resolved in commit ed4e708. The `propose` function has been updated in the `FraxGovernanceOmega` contract to revert whenever it is called. This update will prevent a proposal from being submitted that would interact with a safe that is pending approval to join the allowlist in `FraxGovernanceAlpha`.

TOB-FRAXGOV-2: Vulnerable project dependency
Resolved in commit ed4e708. The vulnerable project dependency cited in this issue has been updated to a version where the vulnerability has been resolved.

TOB-FRAXGOV-3: Replay protection missing in castVoteWithReasonAndParamsBySig
Resolved in commit 00d0b07. The `castVoteWithReasonAndParamsBySig` function has been updated to include a nonce value in the checked signature. If a malicious actor tried to reuse the same signature to cast the targeted users remaining voting weight, the transaction would revert.

TOB-FRAXGOV-4: Ability to lock any user's tokens using deposit_for
Not resolved. Frax Finance provided the following statement about this issue:

> *This issue has been communicated to partners and users; mitigated by avoiding excessive approvals.*

TOB-FRAXGOV-5: The relay function can be used to call critical safe functions
Resolved in commit 00d0b07. The relay function has been updated in the `Governor` contract to revert whenever it is called. Furthermore, the `propose` function has been updated to revert whenever it is called, so the attack vector for this issue has been removed by both of these updates.

TOB-FRAXGOV-6: Votes can be delegated to contracts
Not resolved. Frax Finance provided the following statement about this issue:

> *This behavior is intended.*

TOB-FRAXGOV-7: Lack of public documentation regarding voting power expiry
Not resolved. Frax Finance provided the following statement about this issue:

> *This will be addressed in documentation.*

TOB-FRAXGOV-8: Spamming risk in propose functions
Partially resolved in commit ed4e708. The `propose` function has been updated in the `FraxGovernanceOmega` contract to revert whenever it is called. However, in the `FraxGovernanceAlpha` contract it is still possible for a user to spam the `propose` function

so long as the caller meets the proposal threshold. Frax Finance provided the following statement about this issue:

> *This will be mitigated via minimum proposal voting power requirements configured during deployment.*