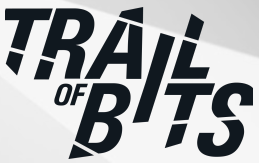

Frax Finance Fraxchain

Security Assessment

February 2, 2024

Prepared for:
Sam Kazemian
Frax Finance

Prepared by: **Bo Henderson, Guillermo Larregay, and Lucas Bourtoule**

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

Project Summary

Contact Information

The following project manager was associated with this project:

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

The following engineering director was associated with this project:

Josselin Feist, Engineering Director, Blockchain
josselin.feist@trailofbits.com

The following consultants were associated with this project:

Bo Henderson, Consultant
bo.henderson@trailofbits.com

Guillermo Larregay, Consultant
guillermo.larregay@trailofbits.com

Lucas Bourtoule, Consultant
lucas.bourtoule@trailofbits.com

Project Timeline

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

Date	Event
December 7, 2023	Pre-project kickoff call
December 15, 2023	Status update meeting #1
December 20, 2023	Delivery of report draft
December 21, 2023	Report readout meeting
February 2, 2024	Delivery of comprehensive report

Executive Summary

Engagement Overview

Frax Finance engaged Trail of Bits to review the security of the Fraxchain contracts. Fraxchain implements a bridge between Ethereum's main network and Frax's own L2 chain, an Optimism fork.

A team of two consultants conducted the review from December 11 to December 21, 2023, for a total of two engineer-weeks of effort. Our testing efforts focused on the Fraxchain Portal, the `VoteEscrowedFXS` (`veFXS`) contract, and the yield-boosting contracts. With full access to source code and documentation, we performed static and dynamic testing of the Fraxchain codebase, using automated and manual processes.

Observations and Impact

Our review of the differences introduced in Fraxchain to the Optimism stack did not uncover any high-severity issues.

The associated tests use both unit and fuzz testing, which provide adequate coverage of some aspects of the business logic. However, mutation testing revealed severe gaps in the test coverage of other components, such as the newly added `frxETH` token's wrapping logic in the `FraxchainPortal` contract and the core business logic of the yield-boosting components. The codebase under audit needs more rigorous unit and fuzz tests.

We found one issue related to error-prone access controls in the `SnapshotDistributor` contract (TOB-FRAXCHAIN-1). Other contracts, such as the `veFXS` contract, feature a more robust two-step ownership transfer process.

The documentation of Fraxchain is sparse, and we often needed to consult external Optimism or Curve documentation for specifications. Thorough public documentation is necessary to ensure users are sufficiently informed to safely interact with this L2 chain.

Recommendations

Based on the codebase maturity evaluation and findings identified during the security review, Trail of Bits recommends that Fraxchain take the following steps prior to deployment:

- **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.
- **Expand the test suite.** The core functionality of the `FraxchainPortal` contract features tests in the `optimism` repository. However, tests covering the integration

of this modified contract with the Optimism stack are lacking. Create a comprehensive test suite that verifies new features and interactions between the portal and other Optimism contracts.

There are also large gaps in the coverage of yield-boosting contracts, and tests for every function and modifier are needed. Run a mutation testing campaign and review the results (see appendix C for details), and add tests that invalidate the resulting mutants.

- **Create an incident response plan and perform regular dry runs.** An incident response plan (see appendix E and these tips on creating such a plan) helps prepare for various failure scenarios. If a real incident occurs, having such a plan will greatly help in reacting to the incident. By performing regular dry runs of the scenarios outlined in the incident response plan, gaps and problems can be identified and fixed and overall improvements and refinements can be made. In addition, it will help train Frax Finance's employees on handling such a situation.
- **Create user-facing documentation for Fraxchain.** Currently there is no user-facing documentation for this project. Without this, it is difficult for users and other integrating parties to understand the protocol, use it, and integrate with it securely.

Finding Severities and Categories

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

EXPOSURE ANALYSIS

Severity	*Count*
High	0
Medium	0
Low	0
Informational	1
Undetermined	0

CATEGORY BREAKDOWN

Category	*Count*
Access Controls	1

Project Goals

The engagement was scoped to provide a security assessment of Fraxchain. Specifically, we sought to answer the following non-exhaustive list of questions:

- Are there any incorrect or error-prone steps in the deployment of the Fraxchain system?
- Could funds become stuck or be stolen during the `frxETH` token's wrapping steps implemented by the `FraxchainPortal` contract?
- Do any changes introduced in the `FraxchainPortal` contract lead to dangerous interactions with the rest of the Optimism stack?
- Is the new vote-weighting system introduced by the `veFXS` contract resistant to manipulation?
- Considering the differences in calculations, is the `veFXS` implementation compatible with the `VoteEscrowedCRV` (veCRV) implementation it was based on?
- Do all functions have appropriate access controls and event emissions?
- Are the access controls and business logic of the `SnapshotDistributor` and `YieldBoosterBridge` contracts correct?

Project Targets

The engagement involved a review and testing of the targets listed below.

dev-fraxchain-contracts

Repository	https://github.com/FraxFinance/dev-fraxchain-contracts
Version	`727a8309a18a58ba435c155f146015588bc30702`
Type	Solidity
Platform	EVM

fragment-vefxs

Repository	https://github.com/FraxFinance/dev-fraxchain-contracts
Version	`07cf3fa4b27fa101d50c879b0ed7646aa8f81208`
Type	Solidity
Platform	EVM
Path	`src/contracts/vefxs/VoteEscrowedFXS.sol`

fraxchain-deployments

Repository	https://github.com/FraxFinance/fraxchain-deployments
Version	`cbb5258134dc669d2a161edebbe5aa398aa30bb1`
Type	Solidity
Platform	EVM
Path	`holesky-testnet/2023-11-14-deploy/script/Deploy.s.sol`

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:

- **`Deploy` script.** This is a Solidity script that coordinates deployment of the Fraxchain stack using Foundry cheat codes to access config files on the host filesystem and to prepare transactions to be broadcast on a live network. We created a flow of bookmarks to facilitate an end-to-end walkthrough of all constructor and initialization logic.

- **`FraxchainPortal` contract.** This is an L1 contract that holds funds native to layer 1 and acts as a low-level entry point for sending messages to the L2 chain. It is a fork of the `OptimismPortal` contract, and we manually reviewed these contracts side by side to analyze the differences, which primarily involve pre-deposit wrapping of ETH to `frxETH`. We traced the flow of funds through the `FraxchainPortal` contract's wrapping process to check whether funds can become stuck or be stolen. We also reviewed the additional modifications to the initialization process to check whether this contract can be safely and correctly deployed.

- **veFXS contract.** This contract is a Solidity rewrite of Curve's veCRV contract but with some modifications, including a different vote-boosting algorithm. We could not implement full differential fuzzing because the results were expected to be different due to the changes in the algorithm, but we reviewed some properties and invariants.

- **`YieldBoosterBridge` contract.** This contract implements the on-chain part of an L1/L2 bridge. It is symmetrically deployed on both chains. When the tokens are mintable, the bridge transfers funds between two chains using a burn-and-mint mechanism. When the tokens are not mintable, the bridge defaults to locking funds on both sides. As the rate between the two tokens changes, the contract transfers the yield in the appropriate direction to compensate. It relies on other contracts from Optimism and OpenZeppelin. We reviewed the business logic of this contract to check whether funds are transferred without error.

- **`SnapshotDistributor` contract.** This contract allows the distribution of a token to registered users according to their share of another snapshot token. We reviewed the access controls implemented by this contract and verified the business logic to assess whether funds are being distributed without error.

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:

- We did not perform an in-depth review of the underlying Optimism smart contracts or the off-chain components. Our review focused on the differences between Fraxchain and Optimism and the additional ancillary contracts introduced by Fraxchain.
- All contracts besides `Deploy.s.sol` and its dependencies were considered explicitly out of scope during our review of the `fraxchain-deployments` repository.
- All contracts besides veFXS and its dependencies were considered explicitly out of scope on the `fragment-vefxs` branch of the `dev-fraxchain-contracts` repository. Other contracts in this repository were reviewed on the development branch at the `727a8309` commit, as specified in the Project Targets section.
- The following smart contracts in the `dev-fraxchain-contracts` repository were considered explicitly out of scope:
  - `src/contracts/L2/FraxchainL1Block.sol`
  - `src/contracts/L2/Multicall3.sol`
  - `src/contracts/L2/ERC20s/sfrxETH.sol`
  - `src/contracts/L2/ERC20s/Frax.sol`
  - `src/contracts/L2/ERC20s/Fpi.sol`
  - `src/contracts/L2/ERC20s/Fpis.sol`
  - `src/contracts/L2/ERC20s/OwnedV2.sol`
  - `src/contracts/L2/ERC20s/Fxs.sol`

Automated Testing

Trail of Bits uses automated techniques to extensively test the security properties of software. We use both open-source static analysis and fuzzing utilities, along with tools developed in house, to perform automated testing of source code and compiled software.

Test Harness Configuration

We used the following tools in the automated testing phase of this project:

Tool	Description	Policy
Slither	A static analysis framework that can statically verify algebraic relationships between Solidity variables	N/A
Echidna	A smart contract fuzzer that can rapidly test security properties via malicious, coverage-guided test case generation	N/A
universalmutator	A deterministic mutation generator that detects gaps in test coverage	Appendix C

Test Results

The results of this focused testing are detailed below.

Echidna. A basic differential fuzzing harness was implemented, with the goal of checking whether the veFXS implementation is compatible with the reference veCRV implementation. Since there are differences in slope and bias calculations and the veFXS implementation allows multiple locks from the same address, the test could not be implemented as a one-to-one feature match between contracts. For the test cases in the harness, a limit of one lock per address was set, and the behavior was compared.

The table below summarizes the tests implemented:

Property	Tool	Result
State-changing functions in both contracts either revert or succeed at the same time. One test was implemented for each state-changing function that was common to both implementations.	Echidna	**Passed**

State changes for common functionality in both contracts are consistent.	Echidna	**Passed**
Fuzzing tests are implemented to fully cover common veFXS functions.	Echidna	**Passed**

universalmutator. The following table displays the proportion of mutants for which all unit tests passed. A small number of valid mutants indicates that test coverage is thorough and that any newly introduced bugs are likely to be caught by the test suite. A large number of valid mutants indicates gaps in the test coverage where errors may go unnoticed. We used the results in the following table to guide our manual review, giving extra attention to the code for which test coverage appears to be incomplete.

Target	Valid Mutants
`dev-fraxchain-contracts/src/contracts/L1/FraxchainPortal.sol`	**20.5%**
`dev-fraxchain-contracts/src/contracts/yieldboosting/ERC4626PriceOracle.sol`	**0.0%**
`dev-fraxchain-contracts/src/contracts/yieldboosting/YieldBoosterBridge.sol`	**19.6%**
`dev-fraxchain-contracts/src/contracts/yieldboosting/SnapshotDistributor.sol`	**36.3%**
`dev-fraxchain-contracts/src/contracts/L2/ERC20s/wfrxETH.sol`	**37.2%**
`dev-fraxchain-contracts/src/contracts/L2/ERC20s/ERC20PermitPermissionedOptiMintable.sol`	**32.9%**

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	Solidity 0.8 is used to protect against underflows, and unchecked arithmetic is used only to increment loop counters. Although the yield-boosting contracts would benefit from more rigorous specifications, the most complex arithmetic is in the `veFXS` contract, which is well specified by existing `veCRV` documentation. Unit and Foundry-based fuzz tests are present and appear to provide good coverage of the system's arithmetic.	**Satisfactory**
Auditing	Events are emitted for all critical functions with sufficient context for off-chain monitoring. Events feature NatSpec-style comments, but they would benefit from additional specifications regarding the types of event information that could indicate unexpected behavior. No incident response plan was provided.	**Moderate**
Authentication / Access Controls	We did not identify any instances of inadequate access controls. Administrative roles generally follow the principle of least privilege. The admin role of the `veFXS` contract features a two-step process for transferring ownership, but the `Proxy` contract inherited from Optimism and the `SnapshotDistributor` contract do not. Better documentation of the privileged actors and their responsibilities would benefit future auditors and developers.	**Satisfactory**

Complexity Management	The `FraxchainPortal` contract follows the patterns established by the `OptimismPortal` contract, and deviations are appropriately placed, although additional inline comments describing these deviations would be beneficial. The internal `_checkpoint` method of the `veFXS` contract features high cyclomatic complexity, but it follows the well-documented patterns established by the `veCRV` contract. There are code repetitions in the `veFXS` contract, mainly for the binary search algorithm. Throughout the in-scope code, including the yield-boosting contracts, functions are well scoped and testable.	**Satisfactory**
Decentralization	The Optimism inheritance leads to critical functionality being upgradable by a single entity without an opportunity for users to opt out. However, privileged roles without the ability to upgrade implementations are limited in their ability to seize user funds. We strongly recommend that the addresses controlling key administrative roles of, for example, proxy contracts, be multisignature contracts that follow key management best practices. Additionally, documentation regarding privileged parties and their trust assumptions would help inform users as they interact with Fraxchain.	**Moderate**
Documentation	Inline documentation, including NatSpec comments, is present and thorough for the most part. High-level documentation is sparse, but large amounts of the specifications available through Optimism and Curve apply to the code under review. The protocol would benefit from additional high-level documentation that clearly describes the differences between the Fraxchain code and the code it was based on.	**Moderate**
Low-Level Manipulation	Although assembly is not used by the in-scope contracts, low-level calls are used sparingly to transfer native tokens to trusted addresses. A `SafeCall` library facilitates low-level interaction with untrusted contracts to guard against illicit gas use.	**Satisfactory**

Testing and Verification	Both unit and fuzz tests are used to verify the correctness of the assessed code. The `FraxchainPortal` and `OptimismPortal` contracts' core functionalities lack sufficient tests, but the originating repositories feature thorough tests. Some novel features lack sufficiently thorough tests (see appendix C for examples and information regarding test coverage assessment). The test coverage of the veFXS contract could not be adequately assessed due to time limitations.	**Moderate**
Transaction Ordering	We did not identify any major front-running risks in the `FraxchainPortal` contract, largely due to its primary role of coordinating with off-chain layer 2 nodes, which are inherently isolated from most front-running risks. We also did not identify any major front-running risks in the veFXS contract.	**Satisfactory**

Summary of Findings

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

ID	Title	Type	Severity
1	Error-prone administrator management	Access Controls	**Informational**

Detailed Findings

1. Error-prone administrator management

Severity: **Informational**	Difficulty: High
Type: Access Controls	Finding ID: TOB-FRAXCHAIN-1
Target: `dev-fraxchain-contracts/src/contracts/yieldboosting/SnapshotDistributor.sol`	

Description

In the `SnapshotDistributor` contract's constructor, the administrator is initialized to the `msg.sender` variable, whose private key is exposed to deployment dependencies on the developer computer. The address assigned to this role can be changed after deployment via an error-prone, one-step process by calling the `setAmin` method.

```
function setAmin(address _newAdmin) external {
    require(msg.sender == admin, "!Auth");
    admin = _newAdmin;
}
```

Figure 1.1: The `setAmin` method (`SnapshotDistributor.sol`)

This method lacks a zero-address check, and if an incorrect `_newAdmin` value is passed, the admin role will be permanently lost.

Exploit Scenario

Alice, a Fraxchain administrator, discovers that one of her deployment dependencies included malicious code. To protect the admin role, she rushes to call `setAmin` and transfer the role to an address that was not exposed to her development environment. She accidentally provides an incorrect address and the admin role is permanently lost. The operator role is an externally owned account (EOA), and later, the private key is lost. As a result, all funds in the `SnapshotDistributor` contract are frozen.

Recommendations

Short term, initialize the admin role to a separate address provided in the constructor to separate deployment and administration concerns. Additionally, implement a two-step process to transfer contract ownership, where the owner proposes a new address and then the new address executes a call to accept the role, completing the transfer.

Long term, identify and document all possible actions that can be taken by privileged accounts, along with their associated risks. This will facilitate reviews of the codebase and prevent future mistakes.

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
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
Transaction Ordering	The system's resistance to transaction-ordering attacks

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. Mutation Testing

This appendix outlines how we conducted mutation testing and highlights some of the most actionable results.

At a high level, mutation tests make several changes to each line of a target file and rerun the test suite for each change. Changes that result in test failures indicate adequate test coverage, while changes that do not cause tests to fail indicate gaps in test coverage. Although mutation testing is a slow process, it allows auditors to focus their review on areas of the codebase that are most likely to contain latent bugs, and it allows developers to identify and add missing tests.

We used universalmutator to conduct our mutation testing campaign. Although this tool is language-agnostic, it provides valuable insights into the unit tests covering this codebase. This tools can be installed with the following command:

```
pip install universalmutator
```

Once installed, a mutation campaign can be run against all Solidity source and test files using the following Bash script:

```
find src/contracts \
  -name '*.sol' \
  -not -path 'interfaces' \
  -print0 | while IFS= read -r -d '' file
do
  log="mutants/$(basename "$file").log"
  mutate "$file" --cmd "timeout 180s forge test --ffi" > ".tmp.log"
  head -n 4 ".tmp.log" > "$log"
  tac ".tmp.log" | sed -n '/.*break;\.\.\.VALID/{N;p;} ;
/.*continue;\.\.\.VALID/{N;p;} ; /.*\.\.\.VALID/p; ' | tac >> "$log"
   tail -n 4 ".tmp.log" >> "$log"
done
```

Figure C.1: A Bash script that runs a mutation testing campaign against each Solidity file in the `src/contracts` directory of the `dev-fraxchain-contracts` repository

Consider the following notes about the above Bash script:

- The overall runtime of the script is approximately four hours on a consumer-grade laptop.
- The `--cmd` argument given to the `mutate` executable (provided by universalmutator) designates the test command. Given the long runtime, a timeout is enforced to prevent any single test from stalling the campaign.

- The `tac` and `sed` commands remove all invalid mutants from the output of `mutate` for quicker and easier analysis of the results. The `head` and `tail` tools are used to preserve the analysis summaries.

An abbreviated, illustrative example of a mutation test output file is shown in figure C.2.

```
*** UNIVERSALMUTATOR ***
MUTATING WITH RULES: .custom-solidity.rules
SKIPPED 859 MUTANTS ONLY CHANGING STRING LITERALS
1353 MUTANTS GENERATED BY RULES
PROCESSING MUTANT: 267: require(_tx.target != FRXETH, "SNIP");  ==>
                        /*require(_tx.target != FRXETH, "SNIP");*/...VALID
PROCESSING MUTANT: 268: if (msg.value > 0) {  ⇒
                        if (false) {...VALID
PROCESSING MUTANT: 270: !isFrxETHTransfer && msg.value <= _tx.value,  ==>
                        (true) && msg.value <= _tx.value,...VALID
PROCESSING MUTANT: 460: require(sent, "SNIP");  ==>
                        /*require(sent, "SNIP");*/...VALID
278 VALID MUTANTS
1075 INVALID MUTANTS
0 REDUNDANT MUTANTS
Valid Percentage: 20.546932742054693%
```

Figure C.2: Abbreviated output from the mutation testing campaign on `FraxchainPortal.sol`

The output has been reformatted slightly (e.g., differences highlighted) for ease of review. In summary, the following lines of `FraxchainPortal.sol` feature incomplete or missing tests:

- Line 267: the requirement that the `frxETH` token not be targeted
- Line 268: the conditional that executes if `msg.value` is nonzero
- Line 270: the guard that applies if the user is not transferring the `frxETH` asset
- Line 460: the requirement that the native token balance be successfully transferred to the minter

We recommend that the Frax Finance team review and write tests that would invalidate these mutants for the `FraxchainPortal` contract, along with, at a bare minimum, all logic added beyond the default `OptimismPortal` contract. Additionally, we recommend that the team add more tests for all the other contracts to cover every conditional statement, `require` statement, and function modifier in the file. Then use a script similar to that provided in figure C.1 to rerun a mutation testing campaign to ensure that the added tests provide adequate coverage.

D. Code Quality Recommendations

The following areas for improvement are not associated with specific vulnerabilities. However, addressing them would enhance code readability and may prevent the introduction of vulnerabilities in the future.

- **Make values set in the constructor immutable.** Specifying such values as immutable will reduce gas costs and make implicit assumptions explicit. The following state variables are set by the constructor and cannot be updated:
  - `SnapshotDistributor.snapshotToken`
  - `SnapshotDistributor.distributeToken`
  - `VoteEscrowedFXS.token`
  - `VoteEscrowedFXS.decimals`
- **Make hard-coded values constant.** Specifying values that are hard coded and never changed as constant will reduce gas costs and make implicit assumptions explicit. The following state variables are initialized to a value and cannot be updated:
  - `wfrxETH.name`
  - `wfrxETH.symbol`
  - `wfrxETH.decimals`
- **Remove unnecessary Boolean comparisons.** A comparison with `true` in a `require` statement is not necessary, and a comparison with `false` can be replaced with a preceding exclamation point [`!`]. The following functions would benefit from such readability enhancements:
  - `FraxchainPortal.whenNotPaused`
  - `FraxchainPortal.finalizeWithdrawalTransaction`
  - `ERC20PermitPermissionedOptiMintable.onlyMinters`
  - `ERC20PermitPermissionedOptiMintable.addMinter`
  - `ERC20PermitPermissionedOptiMintable.removeMinter`

- **Remove unused state variables.** The following state variables are defined but never accessed. To clarify developer intentions, either these variables should be removed, or comments should be added explaining why they are present.
    - `VoteEscrowedFXS.VOTE_WEIGHT_MULTIPLIER_UINT256`
    - `YieldBoosterBridge.RECEIVE_DEFAULT_GAS_LIMIT`
- **Clean temporary variables after they are used.** Some variables are not cleaned and could cause unexpected behavior if they are used again.
    - `futureAdmin` in the veFXS contract: If it Is not cleaned, `futureAdmin` can call the `acceptTransferOwnership` function multiple times, generating excessive unnecessary events.
    - `pending` in the `SnapshotDistributor` contract: After deleting an address, the same pending address could be accidentally reapproved since it is still in the `pending` array.
- **Fix the function name typo in the `SnapshotDistributor` contract.** The `setAmin` function should be named `setAdmin`.
- **Implement a two-step administrator address change in the `SnapshotDistributor` contract.** Since all functionality is controlled by the administrator address, setting it to an incorrect value can make the contract unusable.
- **Replace unsigned value comparisons.** In the veFXS contract, unsigned values are required to be less than or equal to zero. Since an unsigned value cannot be less than zero, the comparison should be only for equality.
- **Extract helper functions.** A binary search algorithm is reimplemented multiple times in the veFXS contract. Identify such instances of repeated logic and create one helper function to improve code readability and maintainability.

E. Incident Response 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.).**
- **Document internal processes for addressing situations in which a deployed remedy does not work or introduces a new bug.**
    - Consider documenting a plan of action for handling failed remediations.
- **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.