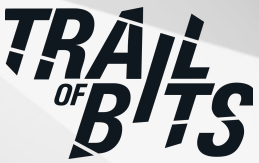

Frax Solidity

Security Assessment

January 10, 2022

Prepared for:
Sam Kazemian
Jason Huan
Travis Moore
Frax Finance

Prepared by:
Samuel Moelius
Maximilian Krüger
Troy Sargent

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 80+ 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

Overview

Frax Finance engaged Trail of Bits to review the security of its smart contracts. From December 6 to December 21, 2021, a team of three consultants conducted a security review of the client-provided source code, with four person-weeks of effort. Details of the project's timeline, test targets, and coverage are provided in subsequent sections of this report.

Project Scope

We focused our testing efforts on the identification of flaws that could result in a compromise or lapse of confidentiality, integrity, or availability of the target system. We performed automated testing and a manual review of the code.

Summary of Findings

The findings concern a broad range of categories. Those of the highest severity concern data validation, timing, and undefined behavior. We attribute these issues to Frax Solidity's rapid expansion. Every exchange or sidechain that the protocol interacts with adds to the attack surface. Hence, we recommend **reducing Frax Solidity's rate of growth and focusing on securing Frax Solidity's existing code**. The following points expand on this recommendation:

- **Develop reproducible tests for all contracts**. The existing tests seem to require substantial effort to run. Ideally, this process should be seamless; for example, a user should be able to clone the Frax Solidity repository and run all tests with a single command. Having reproducible tests is one of the best ways to ensure a codebase's functional correctness.

- **Incorporate testing into the CI pipeline**. Add a branch that contains the exact code of all deployed contracts to the repository. Update this branch whenever a contract is updated or a new contract is deployed. Run all tests on this branch prior to any contract update or new contract deployment. This will help ensure that the proper code is tested.

- **Decommission unneeded or rarely used contracts**. Doing so will reduce the attack surface overall.

- **Expand the documentation**. Frax Solidity has extensive documentation describing the project's goals and philosophy. However, the documentation could be expanded in the following ways:

    - **Document the components and how they interact**. Documentation describing the project's architecture does not appear to exist. Many of the contracts that hold collateral integrate heavily with other projects, and our cursory review of these areas suggest that they are susceptible to common pitfalls and exploits.

    - **Document anticipated behavior**. The game theory, monetary policy, and incentive scheme behind the stablecoin fail to specify what is an exploit and what is anticipated behavior. Furthermore, many of the discretionary actions of the team are not explained or documented. Thus, it is unclear whether the system is robust. The roles and responsibilities of all parties involved (e.g., contract owners, governance, token managers, and custodians) should be clarified.

- **Maintain a list of all the deployed contracts and their commit hashes**. Store this list in the repository so that its history can be reviewed. Update the list whenever a contract is updated or a new contract is deployed. Maintaining this list will facilitate auditing and addressing the preceding bullets related to automating testing, reducing the attack surface, and developing architectural documentation.

- **Consolidate duplicate code**. Identify pieces of code that are semantically similar. Factor out those pieces of code into separate functions where it makes sense to do so. A good example is the role of the "custodian," which many Frax Solidity contracts use. Custodian-related code could be factored out into a single contract from which contracts needing a custodian could inherit.

We recommend that Frax Finance review our Token Integration Checklist in appendix D before interoperating with another system.

After implementing these recommendations, Frax Finance should consider seeking the following services from Trail of Bits in a subsequent review:

- **Review test coverage**. Once reproducible tests have been developed, we can compute their code coverage, identify gaps, and make recommendations for additional tests.

- **Identify invariants and develop tests for them**. Once reproducible tests have been developed, we can identify inputs that could be made arbitrary and write Echidna tests to verify that the code respects the generalized properties.

- **Review the effectiveness of attack surface reduction**. After unneeded or rarely used contracts have been decommissioned, we can review the code paths they exercised and verify that they are no longer used.

Finally, we recommend that Frax finance maintain the following aspects of the project as the team continues to develop the codebase:

- The documentation describing the project's goals and philosophy is extensive.
- The code contains many comments, which often helped us to understand how and why the code worked.
- There are many safeguards implemented, such as the use of custodians and withdrawal caps. If Frax Solidity were to be hacked, these safeguards would help limit the damage caused.
- The Frax Finance team uses a well-known and widely used multisignature wallet (Gnosis Safe).

EXPOSURE ANALYSIS

Severity	*Count*
High	8
Medium	8
Low	1
Informational	6
Undetermined	2

CATEGORY BREAKDOWN

Category	*Count*
Access Controls	3
Configuration	3
Data Validation	6
Patching	1
Testing	2
Timing	2
Undefined Behavior	8

Project Summary

Contact Information

The following managers were associated with this project:

Dan Guido, Account Manager
dan@trailofbits.com

Cara Pearson, Project Manager
cara.pearson@trailofbits.com

The following engineers were associated with this project:

Samuel Moelius, Consultant
sam.moelius@trailofbits.com

Maximilian Krüger, Consultant
max.kruger@trailofbits.com

Troy Sargent, Consultant
troy.sargent@trailofbits.com

Project Timeline

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

Date	Event
December 3, 2021	Pre-project kickoff call
December 13, 2021	Status update meeting #1
December 21, 2021	Delivery of report draft
December 21, 2021	Report readout meeting
January 10, 2022	Delivery of final report

Project Goals

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

- Do contracts that interact with users employ proper access controls?
- Can an unauthorized user mint FRAX/FXS?
- Can one user withdraw another user's tokens/rewards?
- Can token balances between a sidechain and the mainnet fall out of sync?
- Are events used appropriately?
- Could the Frax Solidity contracts withstand a bank run?
- Could an attacker cause FRAX to lose its peg?

Project Targets

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

Frax Finance

Repository	https://github.com/FraxFinance/frax-solidity
Version	`bd40775e283923aa9e32a107abd426430a99835e`
Contracts	`FraxPoolV3.sol`, `veFXS.vy`
Types	Solidity, Vyper
Platform	Ethereum

Frax Finance

Repository	https://github.com/FraxFinance/frax-solidity
Version	`31dd816b03c5598141b5de7b1595453f0fdddb75`
Contracts	`CrossChainCanonical.sol`, `CCFrax1to1AMM.sol`, `FraxUnifiedFarmTemplate.sol`, `StakingRewards.sol`, `FraxGaugeFXSRewardsDistributor.sol`
Type	Solidity
Platform	Ethereum

Frax Finance

Repository	https://github.com/FraxFinance/frax-solidity
Version	`17c8c9f151278e708fd3b4834389dab9b19ac52e`
Contracts	`FraxUniV3Farm_Stable.sol`, `StakingRewardsMultiGauge.sol`, `Convex_AMO_V2.sol`, `FraxAMOMinter.sol`
Type	Solidity
Platform	Ethereum

Frax Finance

Repository	https://github.com/convex-eth/platform
Version	Deployment at `0x7038C406e7e2C9F81571557190d26704bB39B8f3`
Contract	`BasicCvxHolder.sol`
Type	Solidity

Platform	Ethereum

Frax Finance

Repository	https://github.com/FraxFinance/frax-solidity
Version	`3097a527a2844ee8ce2980ef9c7dbd441615f9fa`
Contract	`veFXSYieldDistributorV4.sol`
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 and their results include the following:

- **Compiler warnings**. Compiling the contracts produces 65 warnings. We reviewed each warning to ensure it was innocuous.
- **Slither**. Running Slither over the codebase produced thousands of warnings. Due to the time limitations of the engagement, we focused on the high-severity warnings.
- **Review of contract modifiers**. We reviewed the use of contract modifiers to ensure they were properly used as an access control mechanism.
- **Manual review**. We manually reviewed the 13 contracts (12 Solidity, 1 Vyper) listed under Project Targets.

Coverage Limitations

Because of the time-boxed nature of testing work, it is common to encounter coverage limitations. The following describes the limitations that we encountered during this project:

- We were able to run only some of the tests. This effectively limited us to static analysis and manual review.
- The pattern of adding contracts to an array and looping over them, as `FRAX.globalCollateralValue` does, could easily introduce a vulnerability. We could have more confidence in the contract integrations if the configuration details were more explicit and clear.

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	We identified some arithmetic issues, such as TOB-FRSOL-005. To gain confidence in a system's use of arithmetic, it must be tested, ideally using invariants. We were unable to effectively test the contracts during this engagement.	**Further Investigation Required**
Auditing	Frax Finance states that it uses Tenderly for monitoring, which we recommend. However, Frax Finance's use of Tenderly was not within the scope of this assessment.	**Not Considered**
Authentication / Access Controls	We identified several issues related to access controls, including TOB-FRSOL-014 and TOB-FRSOL-015. As mentioned in the Findings Summary, the roles and responsibilities of all parties should be clarified.	**Moderate**
Complexity Management	The code contains many comments. However, we often found it difficult to attach a logical purpose to a function. Due to duplicate code in the codebase, we had to keep track of subtle differences between functions.	**Moderate**
Cryptography and Key Management	Frax Finance uses a well-known and widely used multisignature wallet (Gnosis Safe). To the best of our knowledge, Frax Finance's key management is secure. However, key management was not within the scope of this assessment.	**Not Considered**
Decentralization	Contract owners have near complete control over the system. While the concept of governance exists, its role is	**Weak**

	unclear.	
Documentation	The documentation describing the project's goals and purpose is extensive. However, the documentation lacks descriptions of the project's architecture (e.g., the components and how they interact) and the roles and responsibilities of the parties involved.	**Moderate**
Front-Running Resistance	We identified some issues related to timing, including TOB-FRSOL-006 and TOB-FRSOL-011.	**Moderate**
Low-Level Manipulation	Assembly use is minimal, and we identified no issues related to it.	**Satisfactory**
Testing and Verification	The project has some tests. However, they do not work "as-is," they require intricate knowledge of the system to enable, and they are not automated (e.g., part of the CI process).	**Weak**

Summary of Findings

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

ID	Title	Type	Severity
1	Testing is not routine	Testing	Informational
2	No clear mapping from contracts to tests	Testing	Informational
3	amoMinterBorrow cannot be paused	Access Controls	Medium
4	Array updates are not constant time	Undefined Behavior	Medium
5	Incorrect calculation of collateral amount in redeemFrax	Undefined Behavior	High
6	spotPriceOHM is vulnerable to manipulation	Timing	High
7	Return values of Chainlink oracle are not validated	Data Validation	Informational
8	Unlimited arbitrage in CCFrax1to1AMM	Undefined Behavior	High
9	Collateral prices are assumed to always be $1	Undefined Behavior	Medium
10	Solidity compiler optimizations can be problematic	Undefined Behavior	Informational
11	Users are unable to limit amount of collateral paid to FraxPoolV3	Timing	Medium
12	Incorrect default price tolerance in CCFrax1to1AMM	Configuration	Low

13	Significant code duplication	Patching	**Informational**
14	StakingRewardsMultiGauge.recoverERC20 allows token managers to steal rewards	Access Controls	**Medium**
15	Convex_AMO_V2 custodian can withdraw rewards	Access Controls	**Medium**
16	FXS1559 documentation is inaccurate	Configuration	**Informational**
17	Univ3LiquidityAMO defaults price of collateral to $1	Configuration	**Medium**
18	calc_withdraw_one_coin is vulnerable to manipulation	Data Validation	**High**
19	Incorrect valuation of LP tokens	Data Validation	**High**
20	Missing check of return value of transfer and transferFrom	Data Validation	**High**
21	A rewards distributor does not exist for each reward token	Undefined Behavior	**Undetermined**
22	minVeFXSForMaxBoost can be manipulated to increase rewards	Data Validation	**Medium**
23	Most collateral is not directly redeemable by depositors	Undefined Behavior	**Undetermined**
24	FRAX.globalCollateralValue counts FRAX as collateral	Undefined Behavior	**High**
25	Setting collateral values manually is error prone	Data Validation	**High**

Detailed Findings

1. Testing is not routine	
Severity: **Informational**	Difficulty: **Undetermined**
Type: Testing	Finding ID: TOB-FRSOL-001
Target: Various	

Description
The Frax Solidity repository does not have reproducible tests that can be run locally. Having reproducible tests is one of the best ways to ensure a codebase's functional correctness. This finding is based on the following events:

- We tried to carry out the instructions in the Frax Solidity README at commit 31dd816. We were unsuccessful.
- We reached out to Frax Finance for assistance. Frax Finance in turn pushed eight additional commits to the Frax Solidity repository (not counting merge commits).
- With these changes, we were able to run some of the tests, but not all of them.

These events suggest that tests require substantial effort to run (as evidenced by the eight additional commits), and that they were not functional at the start of the assessment.

Exploit Scenario
Eve exploits a flaw in a Frax Solidity contract. The flaw would likely have been revealed through unit tests.

Recommendations
Short term, develop reproducible tests that can be run locally for all contracts. A comprehensive set of unit tests will help expose errors, protect against regressions, and provide a sort of documentation to users.

Long term, incorporate unit testing into the CI process:

- Run the tests specific to contract X when a push or pull request affects contract X.
- Run all tests before deploying any new code, including updates to existing contracts.

Automating the testing process will help ensure the tests are run regularly and consistently.

2. No clear mapping from contracts to tests	
Severity: **Informational**	Difficulty: **Undetermined**
Type: Testing	Finding ID: TOB-FRSOL-002
Target: Various	

Description
There are 405 Solidity files within the `contracts` folder[1], but there are only 80 files within the `test` folder[2]. Thus, it is not clear which tests correspond to which contracts.

The number of contracts makes it impractical for a developer to run all tests when working on any one contract. Thus, to test a contract effectively, a developer will need to know which tests are specific to that contract.

Furthermore, as per TOB-FRSOL-001, we recommend that the tests specific to contract X be run when a push or pull request affects contract X. To apply this recommendation, a mapping from the contracts to their relevant tests is needed.

Exploit Scenario
Alice, a Frax Finance developer, makes a change to a Frax Solidity contract. Alice is unable to determine the file that should be used to test the contract and deploys the contract untested. The contract is exploited using a bug that would have been revealed by a test.

Recommendations
Short term, for each contract, produce a list of tests that exercise that contract. If any such list is empty, produce tests for that contract. Having such lists will help facilitate contract testing following a change to it.

Long term, as per TOB-FRSOL-001, incorporate unit testing into the CI process by running the tests specific to contract X when a push or pull request affects contract X. Automating the testing process will help ensure the tests are run regularly and consistently.

[1] `find contracts -name '*.sol' | wc -l`
[2] `find test -type f | wc -l`

3. amoMinterBorrow cannot be paused

Severity: **Medium**	Difficulty: **High**
Type: Access Controls	Finding ID: TOB-FRSOL-003
Target: `FraxPoolV3.sol`	

Description

The `amoMinterBorrow` function does not check for any of the "paused" flags or whether the minter's associated collateral type is enabled. This reduces the `FraxPoolV3` custodian's ability to limit the scope of an attack.

The relevant code appears in figure 3.1. The custodian can set `recollateralizePaused[minter_col_idx]` to `true` if there is a problem with recollateralization, and `collateralEnabled[minter_col_idx]` to `false` if there is a problem with the specific collateral type. However, `amoMinterBorrow` checks for neither of these.

```
    // Bypasses the gassy mint->redeem cycle for AMOs to borrow collateral
    function amoMinterBorrow(uint256 collateral_amount) external onlyAMOMinters {
        // Checks the col_idx of the minter as an additional safety check
        uint256 minter_col_idx = IFraxAMOMinter(msg.sender).col_idx();

        // Transfer
        TransferHelper.safeTransfer(collateral_addresses[minter_col_idx], msg.sender,
collateral_amount);
    }
```

Figure 3.1: contracts/Frax/Pools/FraxPoolV3.sol#L552-L559

Exploit Scenario

Eve discovers and exploits a bug in an AMO contract. The `FraxPoolV3` custodian discovers the attack but is unable to stop it. The `FraxPoolV3` owner is required to disable the AMO contracts. This occurs after significant funds have been lost.

Recommendations

Short term, require `recollateralizePaused[minter_col_idx]` to be `false` and `collateralEnabled[minter_col_idx]` to be `true` for a call to `amoMinterBorrow` to succeed. This will help the `FraxPoolV3` custodian to limit the scope of an attack.

Long term, regularly review all uses of contract modifiers, such as `collateralEnabled`. Doing so will help to expose bugs like the one described here.

4. Array updates are not constant time

Severity: **Medium**	Difficulty: **Medium**
Type: Undefined Behavior	Finding ID: TOB-FRSOL-004
Target: Various	

Description

In several places, arrays are allowed to grow without bound, and those arrays are searched linearly. If an array grows too large and the block gas limit is too low, such a search would fail.

An example appears in figure 4.1. Minters are pushed to but never popped from `minters_array`. When a minter is removed from the array, its entry is searched for and then set to `0`. Note that the cost of such a search is proportional to the searched-for entry's index within the array. Thus, there will eventually be entries that cannot be removed under the current block gas limits because their positions within the array are too large.

```
    function removeMinter(address minter_address) external onlyByOwnGov {
        require(minter_address != address(0), "Zero address detected");
        require(minters[minter_address] == true, "Address nonexistant");

        // Delete from the mapping
        delete minters[minter_address];

        // 'Delete' from the array by setting the address to 0x0
        for (uint i = 0; i < minters_array.length; i++){
            if (minters_array[i] == minter_address) {
                minters_array[i] = address(0); // This will leave a null in the array
and keep the indices the same
                break;
            }
        }

        emit MinterRemoved(minter_address);
    }
```

Figure 4.1: contracts/ERC20/__CROSSCHAIN/CrossChainCanonical.sol#L269-L285

Note that occasionally popping values from `minters_array` is not sufficient to address the issue. An array can be popped from occasionally, yet its size can still be unbounded.

A similar problem exists in `CrossChainCanonical.sol` with respect to `bridge_tokens_array`. This problem appears to exist in many parts of the codebase.

Exploit Scenario

Eve tricks Frax Finance into adding her minter to the `CrosschainCanonical` contract. Frax Finance later decides to remove her minter, but is unable to do so because `minters_array` has grown too large and block gas limits are too low.

Recommendations

Short term, enforce the following policy throughout the codebase: an array's size is bounded, or the array is linearly searched, but never both. Arrays that grow without bound can be updated by moving computations, such as the computation of the index that needs to be updated, off-chain. Alternatively, the code that uses the array could be adjusted to eliminate the need for the array or to instead use a linked list. Adopting these changes will help ensure that the success of critical operations is not dependent on block gas limits.

Long term, incorporate a check for this problematic code pattern into the CI pipeline. In the medium term, such a check might simply involve regular expressions. In the longer term, use Semgrep for Solidity if or when such support becomes stable. This will help to ensure the problem is not reintroduced into the codebase.

5. Incorrect calculation of collateral amount in redeemFrax

Severity: **High**	Difficulty: **High**
Type: Undefined Behavior	Finding ID: TOB-FRSOL-005
Target: `FraxPoolV3.sol`	

Description

The `redeemFrax` function of the `FraxPoolV3` contract multiplies a FRAX amount with the collateral price to calculate the equivalent collateral amount (see the highlights in figure 5.1). This is incorrect. The FRAX amount should be divided by the collateral price instead. Fortunately, in the current deployment of `FraxPoolV3`, only stablecoins are used as collateral, and their price is set to 1 (also see issue TOB-FRSOL-009). This mitigates the issue, as multiplication and division by one are equivalent. If the collateral price were changed to a value different from 1, the exploit scenario described below would become possible, enabling users to steal all collateral from the protocol.

```
if(global_collateral_ratio >= PRICE_PRECISION) {
    // 1-to-1 or overcollateralized
    collat_out = frax_after_fee
                    .mul(collateral_prices[col_idx])
                    .div(10 ** (6 + missing_decimals[col_idx])); // PRICE_PRECISION
+ missing decimals
    fxs_out = 0;
} else if (global_collateral_ratio == 0) {
    // Algorithmic
    fxs_out = frax_after_fee
                    .mul(PRICE_PRECISION)
                    .div(getFXSPrice());
    collat_out = 0;
} else {
    // Fractional
    collat_out = frax_after_fee
                    .mul(global_collateral_ratio)
                    .mul(collateral_prices[col_idx])
                    .div(10 ** (12 + missing_decimals[col_idx])); // PRICE_PRECISION
^2 + missing decimals
    fxs_out = frax_after_fee
                    .mul(PRICE_PRECISION.sub(global_collateral_ratio))
                    .div(getFXSPrice()); // PRICE_PRECISIONS CANCEL OUT
}
```

Figure 5.1: Part of the `redeemFrax` function (`FraxPoolV3.sol#412–433`)

When considering the $price_A$ of an entity A, it is common to think of it as the amount of another entity B that has a value equivalent to 1 A. The unit of measurement of $price_A$ is $\frac{B}{A}$, or Bs per A.

For example, the price of one apple is the number of units of another entity that can be exchanged for one unit of apple. That other entity is usually the local currency. For the US, the price of an apple is the number of US dollars that can be exchanged for an apple: $price_{apple} = \frac{\$}{apple}$.

1. Given a $price_A$ and an amount of A $amount_A$, one can compute the equivalent $amount_B$ through multiplication: $amount_B = amount_A \cdot price_A$.

2. Given a $price_A$ and an amount of B $amount_B$, one can compute the equivalent $amount_A$ through division: $amount_A = amount_B / price_A$.

In short, multiply if the known amount and price refer to the same entity; otherwise, divide.

The `getFraxInCollateral` function correctly follows rule 2 by dividing a FRAX amount by the collateral price to get the equivalent collateral amount (figure 5.2).

```
function getFRAXInCollateral(uint256 col_idx, uint256 frax_amount) public view
returns (uint256) {
    return frax_amount.mul(PRICE_PRECISION).div(10 **
missing_decimals[col_idx]).div(collateral_prices[col_idx]);
}
```

Figure 5.2: The `getFraxInCollateral` function (`FraxPoolV3.sol#242–244`)

Exploit Scenario

A collateral price takes on a value other than 1. This can happen through either a call to `setCollateralPrice` or future modifications that fetch the price from an oracle (also see issue TOB-FRSOL-009). A collateral asset is worth $1,000. Alice mints 1,000 FRAX for 1 unit of collateral. Alice then redeems 1,000 FRAX for 1 million units of collateral ($1000 \cdot 1000$). As a result, Alice has stolen around $1 billion from the protocol. If the calculation were correct, Alice would have redeemed her 1,000 FRAX for 1 unit of collateral ($1000 / 1000$).

Recommendations

Short term, in `FraxPoolV3.redeemFrax`, use the existing `getFraxInCollateral` helper function (figure 5.2) to compute the collateral amount that is equivalent to a given FRAX amount.

Long term, verify that all calculations involving prices use the above rules 1 and 2 correctly.

6. spotPriceOHM is vulnerable to manipulation

Severity: **High**	Difficulty: **High**
Type: Timing	Finding ID: TOB-FRSOL-006
Target: `FraxPoolV3.sol`, `OHM_AMO.sol`, `Frax.sol`	

Description

The `OHM_AMO` contract uses the Uniswap V2 spot price to calculate the value of the collateral that it holds. This price can be manipulated by making a large trade through the `OHM-FRAX` pool. An attacker can manipulate the apparent value of collateral and thereby change the collateralization rate at will. `FraxPoolV3` appears to contain the most funds at risk, but any contract that uses `FRAX.globalCollateralValue` is susceptible to a similar attack. (It looks like `Pool_USDC` has buybacks paused, so it should not be able to burn FXS, at the time of writing.)

```
function spotPriceOHM() public view returns (uint256 frax_per_ohm_raw, uint256
frax_per_ohm) {
    (uint256 reserve0, uint256 reserve1, ) = (UNI_OHM_FRAX_PAIR.getReserves());

    // OHM = token0, FRAX = token1
    frax_per_ohm_raw = reserve1.div(reserve0);
    frax_per_ohm = reserve1.mul(PRICE_PRECISION).div(reserve0.mul(10 **
missing_decimals_ohm));
}
```

Figure 6.1: `old_contracts/Misc_AMOs/OHM_AMO.sol#L174-L180`

`FRAX.globalCollateralValue` loops through `frax_pools_array`, including `OHM_AMO`, and aggregates `collatDollarBalance`. The `collatDollarBalance` for `OHM_AMO` is calculated using `spotPriceOHM` and thus is vulnerable to manipulation.

```
function globalCollateralValue() public view returns (uint256) {
    uint256 total_collateral_value_d18 = 0;

    for (uint i = 0; i < frax_pools_array.length; i++){
        // Exclude null addresses
        if (frax_pools_array[i] != address(0)){
            total_collateral_value_d18 =
total_collateral_value_d18.add(FraxPool(frax_pools_array[i]).collatDollarBalance());
        }

    }
```

```
    return total_collateral_value_d18;
}
```

Figure 6.2: contracts/Frax/Frax.sol#L180-L191

buyBackAvailableCollat returns the amount the protocol will buy back if the aggregate value of collateral appears to back each unit of FRAX with more than is required by the current collateral ratio. Since globalCollateralValue is manipulable, the protocol can be artificially forced into buying (burning) FXS shares and paying out collateral.

```
function buybackAvailableCollat() public view returns (uint256) {
    uint256 total_supply = FRAX.totalSupply();
    uint256 global_collateral_ratio = FRAX.global_collateral_ratio();
    uint256 global_collat_value = FRAX.globalCollateralValue();

    if (global_collateral_ratio > PRICE_PRECISION) global_collateral_ratio =
PRICE_PRECISION; // Handles an overcollateralized contract with CR > 1
    uint256 required_collat_dollar_value_d18 =
(total_supply.mul(global_collateral_ratio)).div(PRICE_PRECISION); // Calculates
collateral needed to back each 1 FRAX with $1 of collateral at current collat ratio

    if (global_collat_value > required_collat_dollar_value_d18) {
        // Get the theoretical buyback amount
        uint256 theoretical_bbk_amt =
global_collat_value.sub(required_collat_dollar_value_d18);

        // See how much has collateral has been issued this hour
        uint256 current_hr_bbk = bbkHourlyCum[curEpochHr()];

        // Account for the throttling
        return comboCalcBbkRct(current_hr_bbk, bbkMaxColE18OutPerHour,
theoretical_bbk_amt);
    }
    else return 0;
}
```

Figure 6.3: contracts/Frax/Pools/FraxPoolV3.sol#L284-L303

buyBackFXS calculates the amount of FXS to burn from the user, calls burn on the FRAXShares contract, and sends the caller an equivalent dollar amount in USDC.

```
function buyBackFxs(uint256 col_idx, uint256 fxs_amount, uint256 col_out_min)
external collateralEnabled(col_idx) returns (uint256 col_out) {
    require(buyBackPaused[col_idx] == false, "Buyback is paused");
    uint256 fxs_price = getFXSPrice();
    uint256 available_excess_collat_dv = buybackAvailableCollat();

    // If the total collateral value is higher than the amount required at the
current collateral ratio then buy back up to the possible FXS with the desired
collateral
    require(available_excess_collat_dv > 0, "Insuf Collat Avail For BBK");
```

```
    // Make sure not to take more than is available
    uint256 fxs_dollar_value_d18 = fxs_amount.mul(fxs_price).div(PRICE_PRECISION);
    require(fxs_dollar_value_d18 <= available_excess_collat_dv, "Insuf Collat Avail
For BBK");

    // Get the equivalent amount of collateral based on the market value of FXS
provided
    uint256 collateral_equivalent_d18 =
fxs_dollar_value_d18.mul(PRICE_PRECISION).div(collateral_prices[col_idx]);
    col_out = collateral_equivalent_d18.div(10 ** missing_decimals[col_idx]); // In
its natural decimals()

    // Subtract the buyback fee
    col_out =
(col_out.mul(PRICE_PRECISION.sub(buyback_fee[col_idx]))).div(PRICE_PRECISION);

    // Check for slippage
    require(col_out >= col_out_min, "Collateral slippage");

    // Take in and burn the FXS, then send out the collateral
    FXS.pool_burn_from(msg.sender, fxs_amount);
    TransferHelper.safeTransfer(collateral_addresses[col_idx], msg.sender, col_out);

    // Increment the outbound collateral, in E18, for that hour
    // Used for buyback throttling
    bbkHourlyCum[curEpochHr()] += collateral_equivalent_d18;
}
```

Figure 6.4: contracts/Frax/Pools/FraxPoolV3.sol#L488-L517

`recollateralize` takes collateral from a user and gives the user an equivalent amount of FXS, including a bonus. Currently, the `bonus_rate` is set to `0`, but a nonzero `bonus_rate` would significantly increase the profitability of an attack.

```
// When the protocol is recollateralizing, we need to give a discount of FXS to hit
the new CR target
// Thus, if the target collateral ratio is higher than the actual value of
collateral, minters get FXS for adding collateral
// This function simply rewards anyone that sends collateral to a pool with the same
amount of FXS + the bonus rate
// Anyone can call this function to recollateralize the protocol and take the extra
FXS value from the bonus rate as an arb opportunity
function recollateralize(uint256 col_idx, uint256 collateral_amount, uint256
fxs_out_min) external collateralEnabled(col_idx) returns (uint256 fxs_out) {
    require(recollateralizePaused[col_idx] == false, "Recollat is paused");
    uint256 collateral_amount_d18 = collateral_amount * (10 **
missing_decimals[col_idx]);
    uint256 fxs_price = getFXSPrice();

    // Get the amount of FXS actually available (accounts for throttling)
    uint256 fxs_actually_available = recollatAvailableFxs();
```

```
    // Calculated the attempted amount of FXS
    fxs_out =
collateral_amount_d18.mul(PRICE_PRECISION.add(bonus_rate).sub(recollat_fee[col_idx])
).div(fxs_price);

    // Make sure there is FXS available
    require(fxs_out <= fxs_actually_available, "Insuf FXS Avail For RCT");

    // Check slippage
    require(fxs_out >= fxs_out_min, "FXS slippage");

    // Don't take in more collateral than the pool ceiling for this token allows
    require(freeCollatBalance(col_idx).add(collateral_amount) <=
pool_ceilings[col_idx], "Pool ceiling");

    // Take in the collateral and pay out the FXS
    TransferHelper.safeTransferFrom(collateral_addresses[col_idx], msg.sender,
address(this), collateral_amount);
    FXS.pool_mint(msg.sender, fxs_out);

    // Increment the outbound FXS, in E18
    // Used for recollat throttling
    rctHourlyCum[curEpochHr()] += fxs_out;
}
```

Figure 6.5: contracts/Frax/Pools/FraxPoolV3.sol#L519-L550

Exploit Scenario

`FraxPoolV3.bonus_rate` is nonzero. Using a flash loan, an attacker buys OHM with FRAX, drastically increasing the spot price of OHM. When `FraxPoolV3.buyBackFXS` is called, the protocol incorrectly determines that FRAX has gained additional collateral. This causes the pool to burn FXS shares and to send the attacker USDC of the equivalent dollar value. The attacker moves the price in the opposite direction and calls `recollateralize` on the pool, receiving and selling newly minted FXS, including a bonus, for profit. This attack can be carried out until the buyback and "recollateralize" hourly cap, currently 200,000 units, is reached.

Recommendations

Short term, take one of the following steps to mitigate this issue:

- Call `FRAX.removePool` and remove `OHM_AMO`. Note, this may cause the protocol to become less collateralized.
- Call `FraxPoolV3.setBbkRctPerHour` and set `bbkMaxColE18OutPerHour` and `rctMaxFxsOutPerHour` to `0`. Calling `toggleMRBR` to pause USDC buybacks and recollateralizations would have the same effect. The implications of this mitigation on the long-term sustainability of the protocol are not clear.

Long term, do not use the spot price to determine collateral value. Instead, use a time-weighted average price (TWAP) or an oracle such as Chainlink. If a TWAP is used, ensure that the underlying pool is highly liquid and not easily manipulated. Additionally, create a rigorous process to onboard collateral since an exploit of this nature could destabilize the system.

References

- samczsun, "So you want to use a price oracle"
- `euler-xyz/uni-v3-twap-manipulation`

7. Return values of the Chainlink oracle are not validated

Severity: **Informational**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-FRSOL-007
Target: `FraxPoolV3.sol`, `ComboOracle.sol`, `FRAXOracleWrapper.sol`, `FXSOracleWrapper.sol`	

Description

The `latestRoundData` function returns a signed integer that is coerced to an unsigned integer without checking that the value is a positive integer. An overflow (e.g., `uint(-1)`) would drastically misrepresent the price and cause unexpected behavior. In addition, `FraxPoolV3` does not validate the completion and recency of the round data, permitting stale price data that does not reflect recent changes.

```
function getFRAXPrice() public view returns (uint256) {
    ( , int price, , , ) = priceFeedFRAXUSD.latestRoundData();
    return uint256(price).mul(PRICE_PRECISION).div(10 **
chainlink_frax_usd_decimals);
}

function getFXSPrice() public view returns (uint256) {
    ( , int price, , , ) = priceFeedFXSUSD.latestRoundData();
    return uint256(price).mul(PRICE_PRECISION).div(10 **
chainlink_fxs_usd_decimals);
}
```

Figure 7.1: `contracts/Frax/Pools/FraxPoolV3.sol#231–239`

An older version of Chainlink's oracle interface has a similar function, `latestAnswer`. When this function is used, the return value should be checked to ensure that it is a positive integer. However, round information does not need to be checked because `latestAnswer` returns only price data.

Recommendations

Short term, add a check to `latestRoundData` and similar functions to verify that values are non-negative before converting them to unsigned integers, and add an invariant that checks that the round has finished and that the price data is from the current round: `require(updatedAt != 0 && answeredInRound == roundID)`.

Long term, define a minimum update threshold and add the following check: `require((block.timestamp - updatedAt <= minThreshold) && (answeredInRound == roundID))`. Furthermore, use consistent interfaces instead of mixing different versions.

References

- Chainlink `AggregatorV3Interface`

8. Unlimited arbitrage in CCFrax1to1AMM

Severity: **High**	Difficulty: **Low**
Type: Undefined Behavior	Finding ID: TOB-FRSOL-008
Target: `CCFrax1to1AMM.sol`	

Description

The `CCFrax1to1AMM` contract implements an automated market maker (AMM) with a constant price and zero slippage. It is a constant sum AMM that maintains the invariant $k = a + b$, where k must remain constant during swaps (ignoring fees) and a and b are the token balances.

Constant sum AMMs are impractical because they are vulnerable to unlimited arbitrage. If the price difference of the AMM's tokens in external markets is large enough, the most profitable arbitrage strategy is to buy the total reserve of the more expensive token from the AMM, leaving the AMM entirely imbalanced.

Other AMMs like Uniswap and Curve prevent unlimited arbitrage by making the price depend on the reserves. This limits profits from arbitrage to a fraction of the total reserves, as the price will eventually reach a point at which the arbitrage opportunity disappears. No such limit exists in the `CCFrax1to1AMM` contract.

While arbitrage opportunities are somewhat limited by the token caps, fees, and gas prices, unlimited arbitrage is always possible once the reserves or the difference between the FRAX price and the token price becomes large enough. While `token_price` swings are limited by the `price_tolerance` parameter, `frax_price` swings are not limited.

Exploit Scenario

The `CCFrax1to1AMM` contract is deployed, and `price_tolerance` is set to 0.05. A token T is whitelisted with a `token_cap` of 100,000 and a `swap_fee` of 0.0004. A user transfers 100,000 FRAX to an AMM. The price of T in an external market becomes 0.995, the minimum at which the AMM allows swaps, and the price of FRAX in an external market becomes 1.005. Alice buys (or takes out a flash loan of) $100,000 worth of T in the external market. Alice swaps all of her T for FRAX with the AMM and then sells all of her FRAX in the external market, making a profit of $960. No FRAX remains in the AMM.

This scenario is conservative, as it assumes a balance of only 100,000 FRAX and a `frax_price` of 1.005. As `frax_price` and the balance increase, the arbitrage profit increases.

Recommendations

Short term, do not deploy `CCFrax1to1AMM` and do not fund any existing deployments with significant amounts. Those funds will be at risk of being drained through arbitrage.

Long term, when providing stablecoin-to-stablecoin liquidity, use a Curve pool or another proven and audited implementation of the stableswap invariant.

9. Collateral prices are assumed to always be $1

Severity: **Medium**	Difficulty: **Medium**
Type: Undefined Behavior	Finding ID: TOB-FRSOL-009
Target: `FraxPoolV3.sol`	

Description
In the `FraxPoolV3` contract, the `setCollateralPrice` function sets collateral prices and stores them in the `collateral_prices` mapping. As of December 13, 2021, collateral prices are set to $1 for all collateral types in the deployed version of the `FraxPoolV3` contract.

Currently, only stablecoins are used as collateral within the Frax Protocol. For those stablecoins, $1 is an appropriate price approximation, at most times. However, when the actual price of the collateral differs enough from $1, users could choose to drain value from the protocol through arbitrage. Conversely, during such price fluctuations, other users who are not aware that `FraxPoolV3` assumes collateral prices are always $1 can receive less value than expected.

Collateral tokens that are not pegged to a specific value, like ETH or WBTC, cannot currently be used safely within `FraxPoolV3`. Their prices are too volatile, and repeatedly calling `setCollateralPrice` is not a feasible solution to keeping their prices up to date.

Exploit Scenario
The price of FEI, one of the stablecoins collateralizing the Frax Protocol, changes to $0.99. Alice, a user, can still mint FRAX/FXS as if the price of FEI were $1. Ignoring fees, Alice can buy 1 million FEI for $990,000, mint 1 million FRAX/FXS with the 1 million FEI, and sell the 1 million FRAX/FXS for $1 million, making $10,000 in the process. As a result, the Frax Protocol loses $10,000.

If the price of FEI changes to $1.01, Bob would expect that he can exchange his 1 million FEI for 1.01 million FRAX/FXS. Since `FraxPoolV3` is not aware of the actual price of FEI, Bob receives only 1 million FRAX/FXS, incurring a 1% loss.

Recommendations
Short term, document the arbitrage opportunities described above. Warn users that they could lose funds if collateral prices differ from $1. Disable the option to set collateral prices to values not equal to $1.

Long term, modify the `FraxPoolV3` contract so that it fetches collateral prices from a price oracle.

10. Solidity compiler optimizations can be problematic

Severity: **Informational**	Difficulty: **Low**
Type: Undefined Behavior	Finding ID: TOB-FRSOL-010
Target: `hardhat.config.js`	

Description
Frax Finance has enabled optional compiler optimizations in Solidity.

There have been several optimization bugs with security implications. Moreover, optimizations are actively being developed. Solidity compiler optimizations are disabled by default, and it is unclear how many contracts in the wild actually use them. Therefore, it is unclear how well they are being tested and exercised.

High-severity security issues due to optimization bugs have occurred in the past. A high-severity bug in the `emscripten`-generated `solc-js` compiler used by Truffle and Remix persisted until late 2018. The fix for this bug was not reported in the Solidity CHANGELOG. Another high-severity optimization bug resulting in incorrect bit shift results was patched in Solidity 0.5.6. More recently, another bug due to the incorrect caching of `keccak256` was reported.

A compiler audit of Solidity from November 2018 concluded that the optional optimizations may not be safe.

It is likely that there are latent bugs related to optimization and that new bugs will be introduced due to future optimizations.

Exploit Scenario
A latent or future bug in Solidity compiler optimizations causes a security vulnerability in the Frax Finance contracts.

Recommendations
Short term, measure the gas savings from optimizations and carefully weigh them against the possibility of an optimization-related bug.

Long term, monitor the development and adoption of Solidity compiler optimizations to assess their maturity.

11. Users are unable to limit the amount of collateral paid to FraxPoolV3

Severity: **Medium**	Difficulty: **Medium**
Type: Timing	Finding ID: TOB-FRSOL-011
Target: `FraxPoolV3.sol`	

Description
The amount of collateral and FXS that is paid by the user in `mintFrax` is dynamically computed from the collateral ratio and price. These parameters can change between transaction creation and transaction execution. Users currently have no way to ensure that the paid amounts are still within acceptable limits at the time of transaction execution.

Exploit Scenario
Alice wants to call `mintFrax`. In the time between when the transaction is broadcast and executed, the global collateral ratio, collateral, and/or FXS prices change in such a way that Alice's minting operation is no longer profitable for her. The minting operation is still executed, and Alice loses funds.

Recommendations
Short term, add the `maxCollateralIn` and `maxFXSIn` parameters to `mintFrax`, enabling users to make the transaction revert if the amount of collateral and FXS that they would have to pay is above acceptable limits.

Long term, always add such limits to give users the ability to prevent unacceptably large input amounts and unacceptably small output amounts when those amounts are dynamically computed.

12. Incorrect default price tolerance in CCFrax1to1AMM

Severity: **Low**	Difficulty: **Low**
Type: Configuration	Finding ID: TOB-FRSOL-012
Target: `CCFrax1to1AMM.sol`	

Description

The `price_tolerance` state variable of the `CCFrax1to1AMM` contract is set to 50,000, which, when using the fixed point scaling factor 10^6, corresponds to 0.05. This is inconsistent with the variable's inline comment, which indicates the number 5,000, corresponding to 0.005. A price tolerance of 0.05 is probably too high and can lead to unacceptable arbitrage activities; this suggests that `price_tolerance` should be set to the value indicated in the code comment.

```
uint256 public price_tolerance = 50000; // E6. 5000 = .995 to 1.005
```

Figure 12.1: The `price_tolerance` state variable (`CCFrax1to1AMM.sol#56`)

Exploit Scenario

This issue exacerbates the exploit scenario presented in issue TOB-FRSOL-008. Given that scenario, but with a price tolerance of 50,000, Alice is able to gain $5459 through arbitrage. A higher price tolerance leads to higher arbitrage profits.

Recommendations

Short term, set the price tolerance to 5,000 both in the code and on the deployed contract.

Long term, ensure that comments are in sync with the code and that constants are correct.

13. Significant code duplication

Severity: **Informational**	Difficulty: **Undetermined**
Type: Patching	Finding ID: TOB-FRSOL-013
Target: Various	

Description

Significant code duplication exists throughout the codebase. Duplicate code can lead to incomplete fixes or inconsistent behavior (e.g., because the code is modified in one location but not in all).

For example, the `FraxUnifiedFarmTemplate.sol` and `StakingRewardsMultiGauge.sol` files both contain a `retroCatchUp` function. As seen in figure 13.1, the functions are almost identical.

```
   // If the period expired, renew it
   function retroCatchUp() internal {
       // Pull in rewards from the
rewards distributor

rewards_distributor.distributeReward(addr
ess(this));

       // Ensure the provided reward
amount is not more than the balance in
the contract.
       // This keeps the reward rate in
the right range, preventing overflows due
to
       // very high values of rewardRate
in the earned and rewardsPerToken
functions;
       // Reward + leftover must be less
than 2^256 / 10^18 to avoid overflow.
       uint256 num_periods_elapsed =
uint256(block.timestamp - periodFinish) /
rewardsDuration; // Floor division to the
nearest period

       // Make sure there are enough
tokens to renew the reward period
       for (uint256 i = 0; i <
rewardTokens.length; i++){
           require((rewardRates(i) *
```

```
   // If the period expired, renew it
   function retroCatchUp() internal {
       // Pull in rewards from the
rewards distributor

rewards_distributor.distributeReward(addr
ess(this));

       // Ensure the provided reward
amount is not more than the balance in
the contract.
       // This keeps the reward rate in
the right range, preventing overflows due
to
       // very high values of rewardRate
in the earned and rewardsPerToken
functions;
       // Reward + leftover must be less
than 2^256 / 10^18 to avoid overflow.
       uint256 num_periods_elapsed =
uint256(block.timestamp.sub(periodFinish)
) / rewardsDuration; // Floor division to
the nearest period

       // Make sure there are enough
tokens to renew the reward period
       for (uint256 i = 0; i <
rewardTokens.length; i++){
```

```
rewardsDuration * (num_periods_elapsed +
1)) <=
ERC20(rewardTokens[i]).balanceOf(address(
this)), string(abi.encodePacked("Not
enough reward tokens available: ",
rewardTokens[i])) );
       }

       // uint256 old_lastUpdateTime =
lastUpdateTime;
       // uint256 new_lastUpdateTime =
block.timestamp;

       // lastUpdateTime = periodFinish;
       periodFinish = periodFinish +
((num_periods_elapsed + 1) *
rewardsDuration);

       // Update the rewards and time
       _updateStoredRewardsAndTime();

       // Update the fraxPerLPStored
       fraxPerLPStored =
fraxPerLPToken();
   }
```

```
require(rewardRates(i).mul(rewardsDuratio
n).mul(num_periods_elapsed + 1) <=
ERC20(rewardTokens[i]).balanceOf(address(
this)), string(abi.encodePacked("Not
enough reward tokens available: ",
rewardTokens[i])) );
       }

       // uint256 old_lastUpdateTime =
lastUpdateTime;
       // uint256 new_lastUpdateTime =
block.timestamp;

       // lastUpdateTime = periodFinish;
       periodFinish =
periodFinish.add((num_periods_elapsed.add
(1)).mul(rewardsDuration));

       _updateStoredRewardsAndTime();

       emit
RewardsPeriodRenewed(address(stakingToken
));
   }
```

Figure 13.1: Left: contracts/Staking/FraxUnifiedFarmTemplate.sol#L463-L490 Right: contracts/Staking/StakingRewardsMultiGauge.sol#L637-L662

Exploit Scenario

Alice, a Frax Finance developer, is asked to fix a bug in the `retroCatchUp` function. Alice updates one instance of the function, but not both. Eve discovers a copy of the function in which the bug is not fixed and exploits the bug.

Recommendations

Short term, perform a comprehensive code review and identify pieces of code that are semantically similar. Factor out those pieces of code into separate functions where it makes sense to do so. This will reduce the risk that those pieces of code diverge after the code is updated.

Long term, adopt code practices that discourage code duplication. Doing so will help to prevent this problem from recurring.

14. StakingRewardsMultiGauge.recoverERC20 allows token managers to steal rewards

Severity: **Medium**	Difficulty: **High**
Type: Access Controls	Finding ID: TOB-FRSOL-014
Target: `StakingRewardsMultiGauge.sol`	

Description

The `recoverERC20` function in the `StakingRewardsMultiGauge` contract allows token managers to steal rewards. This violates conventions established by other Frax Solidity contracts in which `recoverERC20` can be called only by the contract owner.

The relevant code appears in figure 14.1. The `recoverERC20` function checks whether the caller is a token manager and, if so, sends him the requested amount of the token he manages. Convention states that this function should be callable only by the contract owner. Moreover, its purpose is typically to recover tokens unrelated to the contract.

```
   // Added to support recovering LP Rewards and other mistaken tokens from other
systems to be distributed to holders
   function recoverERC20(address tokenAddress, uint256 tokenAmount) external
onlyTknMgrs(tokenAddress) {
        // Check if the desired token is a reward token
        bool isRewardToken = false;
        for (uint256 i = 0; i < rewardTokens.length; i++){
            if (rewardTokens[i] == tokenAddress) {
                isRewardToken = true;
                break;
            }
        }

        // Only the reward managers can take back their reward tokens
        if (isRewardToken && rewardManagers[tokenAddress] == msg.sender){
            ERC20(tokenAddress).transfer(msg.sender, tokenAmount);
            emit Recovered(msg.sender, tokenAddress, tokenAmount);
            return;
        }
```

Figure 14.1: contracts/Staking/StakingRewardsMultiGauge.sol#L798-L814

For comparison, consider the `CCFrax1to1AMM` contract's `recoverERC20` function. It is callable only by the contract owner and specifically disallows transferring tokens used by the contract.

```
   function recoverERC20(address tokenAddress, uint256 tokenAmount) external
onlyByOwner {
        require(!is_swap_token[tokenAddress], "Cannot withdraw swap tokens");

        TransferHelper.safeTransfer(address(tokenAddress), msg.sender, tokenAmount);
   }
```

Figure 14.2:
contracts/Misc_AMOs/__CROSSCHAIN/Moonriver/CCFrax1to1AMM.sol#L340-L344

Exploit Scenario

Eve tricks Frax Finance into making her a token manager for the `StakingRewardsMultiGauge` contract. When the contract's token balance is high, Eve withdraws the tokens and vanishes.

Recommendations

Short term, eliminate the token manager's ability to call `recoverERC20`. This will bring `recoverERC20` in line with established conventions regarding the function's purpose and usage.

Long term, regularly review all uses of contract modifiers, such as `onlyTknMgrs`. Doing so will help to expose bugs like the one described here.

15. Convex_AMO_V2 custodian can withdraw rewards

Severity: **Medium**	Difficulty: **High**
Type: Access Controls	Finding ID: TOB-FRSOL-015
Target: `Convex_AMO_V2.sol`	

Description

The `Convex_AMO_V2` custodian can withdraw rewards. This violates conventions established by other Frax Solidity contracts in which the custodian is only able to pause operations.

The relevant code appears in figure 15.1. The `withdrawRewards` function is callable by the contract owner, governance, or the custodian. This provides significantly more power to the custodian than other contracts in the Frax Solidity repository.

```
   function withdrawRewards(
       uint256 crv_amt,
       uint256 cvx_amt,
       uint256 cvxCRV_amt,
       uint256 fxs_amt
   ) external onlyByOwnGovCust {
       if (crv_amt > 0) TransferHelper.safeTransfer(crv_address, msg.sender,
crv_amt);
       if (cvx_amt > 0) TransferHelper.safeTransfer(address(cvx), msg.sender,
cvx_amt);
       if (cvxCRV_amt > 0) TransferHelper.safeTransfer(cvx_crv_address, msg.sender,
cvxCRV_amt);
       if (fxs_amt > 0) TransferHelper.safeTransfer(fxs_address, msg.sender,
fxs_amt);
   }
```

Figure 15.1: `contracts/Misc_AMOs/Convex_AMO_V2.sol#L425-L435`

Exploit Scenario

Eve tricks Frax Finance into making her the custodian for the `Convex_AMO_V2` contract. When the unclaimed rewards are high, Eve withdraws them and vanishes.

Recommendations

Short term, determine whether the `Convex_AMO_V2` custodian requires the ability to withdraw rewards. If so, document this as a security concern. This will help users to understand the risks associated with depositing funds into the `Convex_AMO_V2` contract.

Long term, implement a mechanism that allows rewards to be distributed without requiring the intervention of an intermediary. Reducing human involvement will increase users' overall confidence in the system.

16. The FXS1559 documentation is inaccurate

Severity: **Informational**	Difficulty: **Undetermined**
Type: Configuration	Finding ID: TOB-FRSOL-016
Target: `FXS1559_AMO_V3.sol`	

Description

The FXS1559 documentation states that excess FRAX tokens are exchanged for FXS tokens, and the FXS tokens are then burned. However, the reality is that those FXS tokens are redistributed to veFXS holders.

More specifically, the documentation states the following:

> "Specifically, every time interval t, FXS1559 calculates the excess value above the CR [collateral ration] and mints FRAX in proportion to the collateral ratio against the value. It then uses the newly minted currency to purchase FXS on FRAX-FXS AMM pairs and burn it."

However, in the `FXS1559_AMO_V3` contract, the number of FXS tokens that are burned is a tunable parameter (see figures 16.1 and 16.2). The parameter defaults to, and is currently, `0` (according to Etherscan).

```
        burn_fraction = 0; // Give all to veFXS initially
```

Figure 16.1: contracts/Misc_AMOs/FXS1559_AMO_V3.sol#L87

```
        // Calculate the amount to burn vs give to the yield distributor
        uint256 amt_to_burn = fxs_received.mul(burn_fraction).div(PRICE_PRECISION);
        uint256 amt_to_yield_distributor = fxs_received.sub(amt_to_burn);

        // Burn some of the FXS
        burnFXS(amt_to_burn);

        // Give the rest to the yield distributor
        FXS.approve(address(yieldDistributor), amt_to_yield_distributor);
        yieldDistributor.notifyRewardAmount(amt_to_yield_distributor);
```

Figure 16.2: contracts/Misc_AMOs/FXS1559_AMO_V3.sol#L159-L168

Exploit Scenario

Frax Finance is publicly shamed for claiming that FXS is deflationary when it is not. Confidence in FRAX declines, and it loses its peg as a result.

Recommendations

Short term, correct the documentation to indicate that some proportion of FXS tokens may be distributed to veFXS holders. This will help users to form correct expectations regarding the operation of the protocol.

Long term, consider whether FXS tokens need to be redistributed. The documentation makes a compelling argument for burning FXS tokens. Adjusting the code to match the documentation might be a better way of resolving this discrepancy.

17. Univ3LiquidityAMO defaults the price of collateral to $1

Severity: **Medium**	Difficulty: **Medium**
Type: Configuration	Finding ID: TOB-FRSOL-017
Target: `UniV3LiquidityAMO_V2_old.sol`, `UniV3LiquidityAMO_V2.sol`, `UniV3LiquidityAMO.sol`	

Description

The Uniswap V3 AMOs default to a price of $1 unless an oracle is set, and it is not clear whether an oracle is or will be set. If the contract lacks an oracle, the contract will return the number of collateral units instead of the price of collateral, meaning that it will value each unit of collateral at $1 instead of the correct price. While this may not be an issue for stablecoins, this pattern is error-prone and unclear. It could introduce errors in the global collateral value of FRAX since the protocol may underestimate (or overestimate) the value of the collateral if the price is above (or below) $1.

`col_bal_e188` is the balance, not the price, of the tokens. When `collatDolarValue` is called without an oracle, the contract falls back to valuing each token at $1.

```
function freeColDolVal() public view returns (uint256) {
    uint256 value_tally_e18 = 0;
    for (uint i = 0; i < collateral_addresses.length; i++){
        ERC20 thisCollateral = ERC20(collateral_addresses[i]);
        uint256 missing_decs = uint256(18).sub(thisCollateral.decimals());
        uint256 col_bal_e18 = thisCollateral.balanceOf(address(this)).mul(10 **
missing_decs);
        uint256 col_usd_value_e18 =
collatDolarValue(oracles[collateral_addresses[i]], col_bal_e18);
        value_tally_e18 = value_tally_e18.add(col_usd_value_e18);
    }
    return value_tally_e18;
}
```

Figure 17.1: contracts/Misc_AMOs/UniV3LiquidityAMO_V2.sol#L161-L171

```
function collatDolarValue(OracleLike oracle, uint256 balance) public view returns
(uint256) {
    if (address(oracle) == address(0)) return balance;
    return balance.mul(oracle.read()).div(1 ether);
}
```

Figure 17.2: contracts/Misc_AMOs/UniV3LiquidityAMO_V2.sol#L174-L177

Exploit Scenario

The value of a collateral token is $0.50. Instead of incentivizing recollateralization, the protocol indicates that it is adequately collateralized (or overcollateralized). However, the price of the collateral token is half the $1 default value, and the protocol needs to respond to the insufficient collateral backing FRAX.

Recommendations

Short term, integrate the Uniswap V3 AMOs properly with an oracle, and remove the hard-coded price assumptions.

Long term, review and test the effect of each pricing function on the global collateral value and ensure that the protocol responds correctly to changes in collateralization.

18. calc_withdraw_one_coin is vulnerable to manipulation

Severity: **High**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-FRSOL-018
Target: `MIM_Convex_AMO.sol`	

Description

The `showAllocations` function determines the amount of collateral in dollars that a contract holds. `calc_withdraw_one_coin` is a Curve AMM function based on the current state of the pool and changes as trades are made through the pool. This spot price can be manipulated using a flash loan or large trade similar to the one described in TOB-FRSOL-006.

```
function showAllocations() public view returns (uint256[10] memory return_arr) {
    // ------------LP Balance------------

    // Free LP
    uint256 lp_owned = (mim3crv_metapool.balanceOf(address(this)));

    // Staked in the vault
    uint256 lp_value_in_vault = MIM3CRVInVault();
    lp_owned = lp_owned.add(lp_value_in_vault);

    // ------------3pool Withdrawable------------
    uint256 mim3crv_supply = mim3crv_metapool.totalSupply();

    uint256 mim_withdrawable = 0;
    uint256 _3pool_withdrawable = 0;
    if (lp_owned > 0) _3pool_withdrawable =
mim3crv_metapool.calc_withdraw_one_coin(lp_owned, 1); // 1: 3pool index
```

Figure 18.1: contracts/Misc_AMOs/MIM_Convex_AMO.sol#L145-160

Exploit Scenario

`MIM_Convex_AMO` is included in `FRAX.globalCollateralValue`, and the `FraxPoolV3.bonus_rate` is nonzero. An attacker manipulates the return value of `calc_withdraw_one_coin`, causing the protocol to undervalue the collateral and reach a less-than-desired collateralization ratio. The attacker then calls `FraxPoolV3.recollateralize`, adds collateral, and sells the newly minted FXS tokens, including a bonus, for profit.

Recommendations

Short term, do not use the Curve AMM spot price to value collateral.

Long term, use an oracle or `get_virtual_price` to reduce the likelihood of manipulation.

References

- Medium, "Economic Attack on Harvest Finance—Deep Dive"

19. Incorrect valuation of LP tokens

Severity: **High**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-FRSOL-019
Target: `FraxUniV3Farm_Volatile.sol`, `SushiSwapLiquidityAMO_ARBI.sol`	

Description

The Frax Protocol uses liquidity pool (LP) tokens as collateral and includes their value in the global collateralization value. In addition to the protocol's incorrect inclusion of FRAX as collateral (see TOB-FRSOL-024), the calculation of the value pool tokens representing Uniswap V2-like and Uniswap V3 positions is inaccurate. As a result, the global collateralization value could be incorrect.

`getAmount0ForLiquidity` (`getAmount1ForLiquidity`) returns the amount, not the value, of token0 (token1) in that price range; the price of FRAX should not be assumed to be $1, for the same reasons outlined in TOB-FRSOL-017.

The `userStakedFrax` helper function uses the metadata of each Uniswap V3 NFT to calculate the collateral value of the underlying tokens. Rather than using the current range, the function calls `getAmount0ForLiquidty` using the range set by a liquidity provider. This suggests that the current price of the assets is within the range set by the liquidity provider, which is not necessarily the case. If the market price is outside the given range, the underlying position will contain 100% of one token rather than a portion of both tokens. Thus, the underlying tokens will not be at a 50% allocation at all times, so this assumption is false. The actual redemption value of the NFT is not the same as what was deposited since the underlying token amounts and prices change with market conditions. In short, the current calculation does not update correctly as the price of assets change, and the global collateral value will be wrong.

```
function userStakedFrax(address account) public view returns (uint256) {
    uint256 frax_tally = 0;
    LockedNFT memory thisNFT;
    for (uint256 i = 0; i < lockedNFTs[account].length; i++) {
        thisNFT = lockedNFTs[account][i];
        uint256 this_liq = thisNFT.liquidity;
        if (this_liq > 0){
            uint160 sqrtRatioAX96 = TickMath.getSqrtRatioAtTick(thisNFT.tick_lower);
            uint160 sqrtRatioBX96 = TickMath.getSqrtRatioAtTick(thisNFT.tick_upper);
            if (frax_is_token0){
                frax_tally =
frax_tally.add(LiquidityAmounts.getAmount0ForLiquidity(sqrtRatioAX96, sqrtRatioBX96,
```

```
uint128(thisNFT.liquidity)));
            }
            else {
                frax_tally =
frax_tally.add(LiquidityAmounts.getAmount1ForLiquidity(sqrtRatioAX96, sqrtRatioBX96,
uint128(thisNFT.liquidity)));
            }
        }
    }

    // In order to avoid excessive gas calculations and the input tokens ratios. 50%
FRAX is assumed
    // If this were Uni V2, it would be akin to reserve0 & reserve1 math
    // There may be a more accurate way to calculate the above...
    return frax_tally.div(2);
}
```

Figure 19.1: contracts/Staking/FraxUniV3Farm_Volatile.sol#L241-L263

In addition, the value of Uniswap V2 LP tokens is calculated incorrectly. The return value of `getReserves` is vulnerable to manipulation, as described in TOB-FRSOL-006. Thus, the value should not be used to price LP tokens, as the value will vary significantly when trades are performed through the given pool. Imprecise fluctuations in the LP tokens' values will result in an inaccurate global collateral value.

```
function lpTokenInfo(address pair_address) public view returns (uint256[4] memory
return_info) {
    // Instantiate the pair
    IUniswapV2Pair the_pair = IUniswapV2Pair(pair_address);

    // Get the reserves
    uint256[] memory reserve_pack = new uint256[](3); // [0] = FRAX, [1] = FXS, [2]
= Collateral
    (uint256 reserve0, uint256 reserve1, ) = (the_pair.getReserves());
    {
        // Get the underlying tokens in the LP
        address token0 = the_pair.token0();
        address token1 = the_pair.token1();

        // Test token0
        if (token0 == canonical_frax_address) reserve_pack[0] = reserve0;
        else if (token0 == canonical_fxs_address) reserve_pack[1] = reserve0;
        else if (token0 == arbi_collateral_address) reserve_pack[2] = reserve0;

        // Test token1
        if (token1 == canonical_frax_address) reserve_pack[0] = reserve1;
        else if (token1 == canonical_fxs_address) reserve_pack[1] = reserve1;
        else if (token1 == arbi_collateral_address) reserve_pack[2] = reserve1;
    }
```

Figure 19.2:
contracts/Misc_AMOs/__CROSSCHAIN/Arbitrum/SushiSwapLiquidityAMO_ARBI.sol #L196-L217

Exploit Scenario

The value of LP positions does not reflect a sharp decline in the market value of the underlying tokens. Rather than incentivizing “recollateralization,” the protocol continues to mint FRAX tokens and causes the true collateralization ratio to fall even further. Although the protocol appears to be solvent, due to incorrect valuations, it is not.

Recommendations

Short term, discontinue the use of LP tokens as collateral since the valuations are inaccurate and misrepresent the amount of collateral backing FRAX.

Long term, use oracles to derive the “fair” value of LP tokens. For Uniswap V2, this means using the constant product to compute the value of the underlying tokens independent of the spot price. For Uniswap V3, this means using oracles to determine the current composition of the underlying tokens that the NFT represents.

References

- Christophe Michel, "Pricing LP tokens | Warp Finance hack"
- Alpha Finance, "Fair Uniswap's LP Token Pricing"

20. Missing check of return value of transfer and transferFrom

Severity: **High**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-FRSOL-020
Target: `TWAMM.sol`	

Description

Some tokens, such as BAT, do not precisely follow the ERC20 specification and will return `false` or fail silently instead of reverting. Because the codebase does not consistently use OpenZeppelin's `SafeERC20` library, the return values of calls to `transfer` and `transferFrom` should be checked. However, return value checks are missing from these calls in many areas of the code, opening the `TWAMM` contract (the time-weighted automated market maker) to severe vulnerabilities.

```
function provideLiquidity(uint256 lpTokenAmount) external {
    require(totalSupply() != 0, 'EC3');

    //execute virtual orders
    longTermOrders.executeVirtualOrdersUntilCurrentBlock(reserveMap);

    //the ratio between the number of underlying tokens and the number of lp tokens
must remain invariant after mint
    uint256 amountAIn = lpTokenAmount * reserveMap[tokenA] / totalSupply();
    uint256 amountBIn = lpTokenAmount * reserveMap[tokenB] / totalSupply();

    ERC20(tokenA).transferFrom(msg.sender, address(this), amountAIn);
    ERC20(tokenB).transferFrom(msg.sender, address(this), amountBIn);
[...]
```

Figure 20.1: `contracts/FPI/TWAMM.sol#L125-136`

Exploit Scenario

Frax deploys the `TWAMM` contract. Pools are created with tokens that do not revert on failure, allowing an attacker to call `provideLiquidity` and mint LP tokens for free; the attacker does not have to deposit funds since the `transferFrom` call fails silently or returns `false`.

Recommendations

Short term, fix the instance described above. Then, fix all instances detected by `slither . --detect unchecked-transfer`.

Long term, review the Token Integration Checklist in appendix D and integrate Slither into the project's CI pipeline to prevent regression and catch new instances proactively.

21. A rewards distributor does not exist for each reward token

Severity: **Undetermined**	Difficulty: **Medium**
Type: Undefined Behavior	Finding ID: TOB-FRSOL-021
Target: `FraxUnifiedFarmTemplate.sol`	

Description

The `FraxUnifiedFarmTemplate` contract's `setGaugeController` function (figure 21.1) has the `onlyTknMgrs` modifier. All other functions with the `onlyTknMgrs` modifier set a value in an array keyed only to the calling token manager's token index. Except for `setGaugeController`, which sets the global `rewards_distributor` state variable, all other functions that set global state variables have the `onlyByOwnGov` modifier. This modifier is stricter than `onlyTknMgrs`, in that it cannot be called by token managers. As a result, any token manager can set the rewards distributor that will be used by all tokens. This exposes the underlying issue: there should be a rewards distributor for each token instead of a single global distributor, and a token manager should be able to set the rewards distributor only for her token.

```
function setGaugeController(address reward_token_address, address
_rewards_distributor_address, address _gauge_controller_address) external
onlyTknMgrs(reward_token_address) {
    gaugeControllers[rewardTokenAddrToIdx[reward_token_address]] =
_gauge_controller_address;
    rewards_distributor =
IFraxGaugeFXSRewardsDistributor(_rewards_distributor_address);
}
```

Figure 21.1: The `setGaugeController` function (`FraxUnifiedFarmTemplate.sol#639–642`)

Exploit Scenario

Reward manager A calls `setGaugeController` to set his rewards distributor. Then, reward manager B calls `setGaugeController` to set his rewards distributor, overwriting the rewards distributor that A set.

Later, sync is called, which in turn calls `retroCatchUp`. As a result, `distributeRewards` is called on B's rewards distributor; however, `distributeRewards` is not called on A's rewards distributor.

Recommendations

Short term, replace the global rewards distributor with an array that is indexed by token index to store rewards distributors, and ensure that the system calls `distributeRewards` on all reward distributors within the `retroCatchUp` function.

Long term, ensure that token managers cannot overwrite each other's settings.

22. minVeFXSForMaxBoost can be manipulated to increase rewards

Severity: **Medium**	Difficulty: **Medium**
Type: Data Validation	Finding ID: TOB-FRSOL-022
Target: `FraxCrossChainFarmSushi.sol`	

Description

`minVeFXSForMaxBoost` is calculated based on the current spot price when a user stakes Uniswap V2 LP tokens. If an attacker manipulates the spot price of the pool prior to staking LP tokens, the reward boost will be skewed upward, thereby increasing the amount of rewards earned. The attacker will earn outsized rewards relative to the amount of liquidity provided.

```
function fraxPerLPToken() public view returns (uint256) {
    // Get the amount of FRAX 'inside' of the lp tokens
    uint256 frax_per_lp_token;

    // Uniswap V2
    // ====================================================
    {
        uint256 total_frax_reserves;
        (uint256 reserve0, uint256 reserve1, ) = (stakingToken.getReserves());
[...]
```

Figure 22.1: contracts/Staking/FraxCrossChainFarmSushi.sol#L242-L250

```
function userStakedFrax(address account) public view returns (uint256) {
    return (fraxPerLPToken()).mul(_locked_liquidity[account]).div(1e18);
}

function minVeFXSForMaxBoost(address account) public view returns (uint256) {
    return
(userStakedFrax(account)).mul(vefxs_per_frax_for_max_boost).div(MULTIPLIER_PRECISION
);
}

function veFXSMultiplier(address account) public view returns (uint256) {
    if (address(veFXS) != address(0)){
        // The claimer gets a boost depending on amount of veFXS they have relative
to the amount of FRAX 'inside'
        // of their locked LP tokens
        uint256 veFXS_needed_for_max_boost = minVeFXSForMaxBoost(account);
[...]
```

Figure 22.2: `contracts/Staking/FraxCrossChainFarmSushi.sol#L260-L272`

Exploit Scenario

An attacker sells a large amount of FRAX through the incentivized Uniswap V2 pool, increasing the amount of FRAX in the reserve. In the same transaction, the attacker calls `stakeLocked` and deposits LP tokens. The attacker's reward boost, `new_vefxs_multiplier`, increases due to the large trade, giving the attacker outsized rewards. The attacker then swaps his tokens back through the pool to prevent losses.

Recommendations

Short term, do not use the Uniswap spot price to calculate reward boosts.

Long term, use canonical and audited rewards contracts for Uniswap V2 liquidity mining, such as MasterChef.

23. Most collateral is not directly redeemable by depositors

Severity: **Undetermined**	Difficulty: **Medium**
Type: Undefined Behavior	Finding ID: TOB-FRSOL-023
Target: `FraxPoolV3.sol`, `ConvexAMO.sol`	

Description
The following describes the on-chain situation on December 20, 2021.

The Frax stablecoin has a total supply of 1.5 billion FRAX. Anyone can mint new FRAX tokens by calling `FraxPoolV3.mintFrax` and paying the necessary amount of collateral and FXS. Conversely, anyone can redeem his or her FRAX for collateral and FXS by calling `FraxPoolV3.redeemFrax`. However, the Frax team manually moves collateral from the `FraxPoolV3` contract into AMO contracts in which the collateral is used to generate yield. As a result, only $5 million (0.43%) of the collateral backing FRAX remains in the `FraxPoolV3` contract and is available for redemption. If those $5 million are redeemed, the Frax Finance team would have to manually move collateral from the AMOs to `FraxPoolV3` to make further redemptions possible.

Currently, $746 million (64%) of the collateral backing FRAX is managed by the `ConvexAMO` contract. FRAX owners cannot access the `ConvexAMO` contract, as all of its operations can be executed only by the Frax team.

Exploit Scenario
Owners of FRAX want to use the `FraxPoolV3` contract's `redeemFrax` function to redeem more than $5 million worth of FRAX for the corresponding amount of collateral. The redemption fails, as only $5 million worth of USDC is in the `FraxPoolV3` contract. From the redeemers' perspectives, FRAX is no longer exchangeable into something worth $1, removing the base for its stable price.

Recommendations
Short term, deposit more FRAX into the `FraxPoolV3` contract so that the protocol can support a larger volume of redemptions without requiring manual intervention by the Frax team.

Long term, implement a mechanism whereby the pools can retrieve FRAX that is locked in AMOs to pay out redemptions.

24. FRAX.globalCollateralValue counts FRAX as collateral

Severity: **High**	Difficulty: **Low**
Type: Undefined Behavior	Finding ID: TOB-FRSOL-024
Target: `Frax.sol`	

Description

Each unit of FRAX represents $1 multiplied by the collateralization ratio of debt. That is, if the collateralization ratio is 86%, the Frax Protocol owes each holder of FRAX $0.86. Instead of accounting for this as a liability, the protocol includes this debt as an asset backing FRAX. In other words, FRAX is backed in part by FRAX. Because the `FRAX.globalCollateralValue` includes FRAX as an asset and not debt, the true collateralization ratio is lower than stated, and users cannot redeem FRAX for the underlying collateral in mass for reasons beyond those described in TOB-FRSOL-023. This issue occurs extensively throughout the code.

For instance, the amount FRAX in a Uniswap V3 liquidity position is included in the contract's collateral value.

```
function TotalLiquidityFrax() public view returns (uint256) {
    uint256 frax_tally = 0;
    Position memory thisPosition;
    for (uint256 i = 0; i < positions_array.length; i++) {
        thisPosition = positions_array[i];
        uint128 this_liq = thisPosition.liquidity;
        if (this_liq > 0){
            uint160 sqrtRatioAX96 =
TickMath.getSqrtRatioAtTick(thisPosition.tickLower);
            uint160 sqrtRatioBX96 =
TickMath.getSqrtRatioAtTick(thisPosition.tickUpper);
            if (thisPosition.collateral_address >
0x853d955aCEf822Db058eb8505911ED77F175b99e){ // if address(FRAX) <
collateral_address, then FRAX is token0
                frax_tally =
frax_tally.add(LiquidityAmounts.getAmount0ForLiquidity(sqrtRatioAX96, sqrtRatioBX96,
this_liq));
            }
            else {
                frax_tally =
frax_tally.add(LiquidityAmounts.getAmount1ForLiquidity(sqrtRatioAX96, sqrtRatioBX96,
this_liq));
            }
        }
    }
```

Figure 24.1: contracts/Misc_AMOs/UniV3LiquidityAMO_V2.sol#L199-L216

In another instance, the value of FRAX in FRAX/token liquidity positions on Arbitrum is counted as collateral. Again, FRAX should be counted as debt and not collateral.

```
function lpTokenInfo(address pair_address) public view returns (uint256[4] memory
return_info) {
    // Instantiate the pair
    IUniswapV2Pair the_pair = IUniswapV2Pair(pair_address);

    // Get the reserves
    uint256[] memory reserve_pack = new uint256[](3); // [0] = FRAX, [1] = FXS, [2]
= Collateral
    (uint256 reserve0, uint256 reserve1, ) = (the_pair.getReserves());
    {
        // Get the underlying tokens in the LP
        address token0 = the_pair.token0();
        address token1 = the_pair.token1();

        // Test token0
        if (token0 == canonical_frax_address) reserve_pack[0] = reserve0;
        else if (token0 == canonical_fxs_address) reserve_pack[1] = reserve0;
        else if (token0 == arbi_collateral_address) reserve_pack[2] = reserve0;

        // Test token1
        if (token1 == canonical_frax_address) reserve_pack[0] = reserve1;
        else if (token1 == canonical_fxs_address) reserve_pack[1] = reserve1;
        else if (token1 == arbi_collateral_address) reserve_pack[2] = reserve1;
    }

    // Get the token rates
    return_info[0] = (reserve_pack[0] * 1e18) / (the_pair.totalSupply());
    return_info[1] = (reserve_pack[1] * 1e18) / (the_pair.totalSupply());
    return_info[2] = (reserve_pack[2] * 1e18) / (the_pair.totalSupply());

    // Set the pair type (used later)
    if (return_info[0] > 0 && return_info[1] == 0) return_info[3] = 0; // FRAX/XYZ
    else if (return_info[0] == 0 && return_info[1] > 0) return_info[3] = 1; //
FXS/XYZ
    else if (return_info[0] > 0 && return_info[1] > 0) return_info[3] = 2; //
FRAX/FXS
    else revert("Invalid pair");
}
```

Figure 24.2: contracts/Misc_AMOs/__CROSSCHAIN/Arbitrum/SushiSwapLiquidityAMO_ARBI.sol#L196-L229

Exploit Scenario

Users attempt to redeem FRAX for USDC, but the collateral backing FRAX is, in part, FRAX itself, and not enough collateral is available for redemption. The collateralization ratio does

not accurately reflect when the protocol is insolvent. That is, it indicates that FRAX is fully collateralized in the scenario in which 100% of FRAX is backed by FRAX.

Recommendations

Short term, revise `FRAX.globalCollateralValue` so that it does not count FRAX as collateral, and ensure that the protocol deposits the necessary amount of collateral to ensure the collateralization ratio is reached.

Long term, after fixing this issue, continue reviewing how the protocol accounts for collateral and ensure the design is sound.

25. Setting collateral values manually is error-prone

Severity: **High**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-FRSOL-025
Target: `ManualTokenTrackerAMO.sol`	

Description

During the audit, the Frax Solidity team indicated that collateral located on non-mainnet chains is included in `FRAX.globalCollateralValue` in `FRAXStablecoin`, the Ethereum mainnet contract. (As indicated in TOB-FRSOL-023, this collateral cannot currently be redeemed by users.) Using a script, the team aggregates collateral prices from across multiple chains and contracts and then posts that data to `ManualTokenTrackerAMO` by calling `setDollarBalances`. Since we did not have the opportunity to review the script and these contracts were out of scope, we cannot speak to the security of this area of the system. Other issues with collateral accounting and pricing indicate that this process needs review. Furthermore, considering the following issues, this privileged role and architecture significantly increases the attack surface of the protocol and the likelihood of a hazard:

- The correctness of the script used to calculate the data has not been reviewed, and users cannot audit or verify this data for themselves.
- The configuration of the Frax Protocol is highly complex, and we are not aware of how these interactions are tracked. It is possible that collateral can be mistakenly counted more than once or not at all.
- The reliability of the script and the frequency with which it is run is unknown. In times of market volatility, it is not clear whether the script will function as anticipated and be able to post updates to the mainnet.
- This role is not explained in the documentation or contracts, and it is not clear what guarantees users have regarding the collateralization of FRAX (i.e., what is included and updated).

As of December 20, 2021, `collatDollarBalance` has not been updated since November 13, 2021, and is equivalent to `fraxDollarBalanceStored`. This indicates that `FRAX.globalCollateralValue` is both out of date and incorrectly counts FRAX as collateral (see TOB-FRSOL-024).

Recommendations

Short term, include only collateral that can be valued natively on the Ethereum mainnet and do not include collateral that cannot be redeemed in `FRAX.globalCollateralValue`.

Long term, document and follow rigorous processes that limit risk and provide confidence to users.

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. Non-Security-Related Findings

The following recommendations are not associated with specific vulnerabilities. However, they enhance code readability and may prevent the introduction of vulnerabilities in the future.

General

- In every location in which `MULTIPLIER_PRECISION` is defined, it is defined as `1e18`. However, `MULTIPLIER_PRECISION` is not used consistently in place of `1e18`. Examples of this issue can be found in `FraxUniV3Farm_Stable.sol` on lines 69 and 436:

```
   uint256 private constant MULTIPLIER_PRECISION = 1e18;
```

```
            rwd_rate = (gauge_controller.global_emission_rate())
.mul(last_gauge_relative_weight).div(1e18);
```

- The terms “curator” and “custodian” seem to be used interchangeably, though custodian seems to be more prevalent. The following contracts use the term “curator” outside of a comment:
  - `contracts/Curve/FraxCrossChainRewarder.sol`
  - `contracts/Curve/FraxGaugeFXSRewardsDistributor.sol`
  - `contracts/Curve/IFraxGaugeFXSRewardsDistributor.sol`
  - `contracts/Staking/FraxUniV3Farm_Stable.sol`

`package.json`

- The npm `test` script calls itself recursively:

```
"scripts": {
  "test": "npm test",
  "tsc": "tsc"
},
```

  We recommend that the script be adjusted so that it executes all Hardhat tests. (See also TOB-FRSOL-001.)

`FraxPoolV3.sol`

- Almost everywhere in `FraxPoolV3.sol`, “recollateralize” is referred to before “buyBack.” But in the `toggleMRBR` function, they are reversed:

```
    function toggleMRBR(uint256 col_idx, uint8 tog_idx) external
onlyByOwnGovCust {
        if (tog_idx == 0) mintPaused[col_idx] = !mintPaused[col_idx];
        else if (tog_idx == 1) redeemPaused[col_idx] = !redeemPaused[col_idx];
        else if (tog_idx == 2) buyBackPaused[col_idx] = !buyBackPaused[col_idx];
        else if (tog_idx == 3) recollateralizePaused[col_idx] =
!recollateralizePaused[col_idx];

        emit MRBRToggled(col_idx, tog_idx);
    }
```

We recommend ordering these notions consistently throughout `FraxPoolV3.sol`.

`StakingRewardsMultiGauge.sol`

- The following error message in `StakingRewardsMultiGauge.sol` is incorrect:

```
        require(_vefxs_per_frax_for_max_boost > 0, "veFXS pct max must be >=
0");
```

- Consider adding the following check to `setLockedStakeTimeForMinAndMaxMultiplier`: `lock_time_min <= lock_time_for_max_multiplier`.

```
    function setLockedStakeTimeForMinAndMaxMultiplier(uint256
_lock_time_for_max_multiplier, uint256 _lock_time_min) external onlyByOwner {
        require(_lock_time_for_max_multiplier >= 1, "Mul max time must be >=
1");
        require(_lock_time_min >= 1, "Mul min time must be >= 1");
```

This condition is ensured elsewhere, but adding a check to `setLockedStakeTimeForMinAndMaxMultiplier` will help ensure that the error is caught early.

`Convex_AMO_V2.sol`

- In several places, numeric literals are used where using defined constants would make the code more readable. The following is an example:

```
    function dollarBalances() public view returns (uint256 frax_val_e18, uint256
collat_val_e18) {
        // Get the allocations
        uint256[11] memory allocations = showAllocations();

        frax_val_e18 = (allocations[2]).add((allocations[3]).mul((10 **
missing_decimals)));
        collat_val_e18 = (allocations[6]).mul(10 ** missing_decimals);
    }
```

D. Token Integration Checklist

The following checklist provides recommendations for interactions with arbitrary tokens. Every unchecked item should be justified, and its associated risks, understood. For an up-to-date version of the checklist, see `crytic/building-secure-contracts`.

For convenience, all Slither utilities can be run directly on a token address, such as the following:

```
slither-check-erc 0xdac17f958d2ee523a2206206994597c13d831ec7 TetherToken --erc erc20
slither-check-erc 0x06012c8cf97BEaD5deAe237070F9587f8E7A266d KittyCore --erc erc721
```

To follow this checklist, use the below output from Slither for the token:

```
slither-check-erc [target] [contractName] [optional: --erc ERC_NUMBER]
slither [target] --print human-summary
slither [target] --print contract-summary
slither-prop . --contract ContractName # requires configuration, and use of Echidna
and Manticore
```

General Considerations

- ❏ **The contract has a security review.** Avoid interacting with contracts that lack a security review. Check the length of the assessment (i.e., the level of effort), the reputation of the security firm, and the number and severity of the findings.
- ❏ **You have contacted the developers.** You may need to alert their team to an incident. Look for appropriate contacts on `blockchain-security-contacts`.
- ❏ **They have a security mailing list for critical announcements.** Their team should advise users (like you!) when critical issues are found or when upgrades occur.

Contract Composition

- ❏ **The contract avoids unnecessary complexity.** The token should be a simple contract; a token with complex code requires a higher standard of review. Use Slither's `human-summary` printer to identify complex code.
- ❏ **The contract uses `SafeMath`.** Contracts that do not use `SafeMath` require a higher standard of review. Inspect the contract by hand for `SafeMath` usage.
- ❏ **The contract has only a few non-token-related functions.** Non-token-related functions increase the likelihood of an issue in the contract. Use Slither's `contract-summary` printer to broadly review the code used in the contract.

- ❏ **The token has only one address.** Tokens with multiple entry points for balance updates can break internal bookkeeping based on the address (e.g., `balances[token_address][msg.sender]` may not reflect the actual balance).

Owner Privileges

- ❏ **The token is not upgradeable.** Upgradeable contracts may change their rules over time. Use Slither's `human-summary` printer to determine whether the contract is upgradeable.
- ❏ **The owner has limited minting capabilities.** Malicious or compromised owners can abuse minting capabilities. Use Slither's `human-summary` printer to review minting capabilities, and consider manually reviewing the code.
- ❏ **The token is not pausable.** Malicious or compromised owners can trap contracts relying on pausable tokens. Identify pausable code by hand.
- ❏ **The owner cannot blacklist the contract.** Malicious or compromised owners can trap contracts relying on tokens with a blacklist. Identify blacklisting features by hand.
- ❏ **The team behind the token is known and can be held responsible for abuse.** Contracts with anonymous development teams or teams that reside in legal shelters require a higher standard of review.

ERC20 Tokens

ERC20 Conformity Checks

Slither includes a utility, `slither-check-erc`, that reviews the conformance of a token to many related ERC standards. Use `slither-check-erc` to review the following:

- ❏ **`Transfer` and `transferFrom` return a boolean.** Several tokens do not return a boolean on these functions. As a result, their calls in the contract might fail.
- ❏ **The `name`, `decimals`, and `symbol` functions are present if used.** These functions are optional in the ERC20 standard and may not be present.
- ❏ **`Decimals` returns a `uint8`.** Several tokens incorrectly return a `uint256`. In such cases, ensure that the value returned is below 255.
- ❏ **The token mitigates the known ERC20 race condition.** The ERC20 standard has a known ERC20 race condition that must be mitigated to prevent attackers from stealing tokens.

Slither includes a utility, `slither-prop`, that generates unit tests and security properties that can discover many common ERC flaws. Use `slither-prop` to review the following:

- ❏ **The contract passes all unit tests and security properties from `slither-prop`.** Run the generated unit tests and then check the properties with Echidna and Manticore.

Risks of ERC20 Extensions

The behavior of certain contracts may differ from the original ERC specification. Conduct a manual review of the following conditions:

- ❏ **The token is not an ERC777 token and has no external function call in `transfer` or `transferFrom`.** External calls in the transfer functions can lead to reentrancies.
- ❏ **`Transfer` and `transferFrom` should not take a fee.** Deflationary tokens can lead to unexpected behavior.
- ❏ **Potential interest earned from the token is taken into account.** Some tokens distribute interest to token holders. This interest may be trapped in the contract if not taken into account.

Token Scarcity

Reviews of token scarcity issues must be executed manually. Check for the following conditions:

- ❏ **The supply is owned by more than a few users.** If a few users own most of the tokens, they can influence operations based on the tokens' repartition.
- ❏ **The total supply is sufficient.** Tokens with a low total supply can be easily manipulated.
- ❏ **The tokens are located in more than a few exchanges.** If all the tokens are in one exchange, a compromise of the exchange could compromise the contract relying on the token.
- ❏ **Users understand the risks associated with a large amount of funds or flash loans.** Contracts relying on the token balance must account for attackers with a large amount of funds or attacks executed through flash loans.
- ❏ **The token does not allow flash minting.** Flash minting can lead to substantial swings in the balance and the total supply, which necessitate strict and comprehensive overflow checks in the operation of the token.

ERC721 Tokens

ERC721 Conformity Checks

The behavior of certain contracts may differ from the original ERC specification. Conduct a manual review of the following conditions:

- ❏ **Transfers of tokens to the `0x0` address revert.** Several tokens allow transfers to `0x0` and consider tokens transferred to that address to have been burned; however, the ERC721 standard requires that such transfers revert.
- ❏ **`safeTransferFrom` functions are implemented with the correct signature.** Several token contracts do not implement these functions. A transfer of NFTs to one of those contracts can result in a loss of assets.
- ❏ **The `name`, `decimals`, and `symbol` functions are present if used.** These functions are optional in the ERC721 standard and may not be present.
- ❏ **If it is used, `decimals` returns a `uint8(0)`.** Other values are invalid.
- ❏ **The `name` and `symbol` functions can return an empty string.** This behavior is allowed by the standard.
- ❏ **The `ownerOf` function reverts if the `tokenId` is invalid or is set to a token that has already been burned.** The function cannot return `0x0`. This behavior is required by the standard, but it is not always properly implemented.
- ❏ **A transfer of an NFT clears its approvals.** This is required by the standard.
- ❏ **The token ID of an NFT cannot be changed during its lifetime.** This is required by the standard.

Common Risks of the ERC721 Standard

To mitigate the risks associated with ERC721 contracts, conduct a manual review of the following conditions:

- ❏ **The `onERC721Received` callback is taken into account.** External calls in the transfer functions can lead to reentrancies, especially when the callback is not explicit (e.g., in `safeMint` calls).

- ❏ **When an NFT is minted, it is safely transferred to a smart contract.** If there is a minting function, it should behave similarly to `safeTransferFrom` and properly handle the minting of new tokens to a smart contract. This will prevent a loss of assets.
- ❏ **The burning of a token clears its approvals.** If there is a burning function, it should clear the token's previous approvals.