Frax Finance

Security Assessment

June 11, 2021

Prepared For:
Jason Huan | *Frax Finance*
jason@frax.finance

Travis Moore | *Frax Finance*
travis@frax.finance

Sam Kazemian | *Frax Finance*
sam@frax.finance

Prepared By:
Maximilian Krüger | *Trail of Bits*
max.kruger@trailofbits.com

Spencer Michaels | *Trail of Bits*
spencer.michaels@trailofbits.com

Natalie Chin | *Trail of Bits*
natalie.chin@trailofbits.com

Changelog:

June 1, 2021:	Initial report delivered
June 11, 2021:	Fix log added (Appendix F)

Executive Summary

From May 10 to May 21, 2021, Frax Finance engaged Trail of Bits to review the security of its stablecoin contracts. Trail of Bits conducted this assessment over four person-weeks, with two engineers working from commit `3f0993` and then from commit `6e0352` of the `FraxFinance/frax-solidity` repository.

In addition to conducting a manual review, we used our static analysis tool Slither to answer various questions about the security of the system. We focused on flaws that could lead to the following:

- Riskless arbitrage
- Contract state modification via a bypass of access controls
- Exploits arising from the use of flash loans

During the first week of the assessment, we focused on the `FRAXStablecoin`, `FRAXShares`, `veFXS`, and `Pool_USDC (FraxPool)` contracts. During the second week, we concluded our static analysis and manual review of those contracts as well as the `CurveAMO_V3` contract. We also achieved partial coverage of the `InvestorAMO_V2`.

Our review resulted in 27 findings ranging from high to informational severity. The most severe issues relate to a lack of return value checks on `transfer` and `transferFrom` functions throughout the codebase, which could result in a loss of user funds. Other high-severity issues stem from the use of Aragon's voting contract, which has known problems; a lack of two-step processes for critical operations; and the lack of an existence check for `delegatecall`. The informational-severity issues include the use of Solidity compiler optimizations, as well as a lack of sufficient data validation checks and events for critical operations.

In addition to the security findings, we identified code quality issues not related to any particular vulnerability, which are discussed in Appendix C.

Appendix D contains recommendations on interactions with arbitrary ERC20 tokens. We suggest following these recommendations when integrating additional tokens and pools into the system. Risks associated with configuring `CurveAMO_V3` as publicly callable are discussed in Appendix E.

The FRAX system includes many components and integrations with other protocols such as Aragon, Yearn, and Curve. It has a large attack surface, which exposes the system to increased risk. While the code is reasonably well structured and includes a satisfactory number of unit tests, we found many areas that could be improved, such as missing events and monitoring mechanisms as well as data validation issues. The Frax Finance team should also adhere to smart contract development best practices; these include keeping dependencies up to date, using a build system, implementing test coverage reporting, preventing the duplication of code across repositories, and ensuring the clear versioning of deployed contracts.

Trail of Bits recommends addressing the findings in this report, expanding the unit testing suite to cover more exception paths, determining system invariants, and using fuzzing to check system invariants and critical arithmetic. We also suggest performing a security assessment of protocol components omitted from this review. Lastly, Frax Finance should consider reducing the scope of the project and its integrations with other protocols, including by removing non-essential contracts and protocol integrations that increase the attack surface.

June 11, 2021 Update: Trail of Bits reviewed fixes implemented for the issues in this report. See the results of this fix review in Appendix F: Fix Log.

Project Dashboard

Application Summary

Name	Frax Finance
Versions	`3f0993` and `6e0352`
Type	Solidity
Platform	Ethereum

Engagement Summary

Dates	May 10–May 21, 2021
Method	Full knowledge
Consultants Engaged	2
Level of Effort	4 person-weeks

Vulnerability Summary

Total High-Severity Issues	7	■ ■ ■ ■ ■ ■ ■
Total Medium-Severity Issues	0	
Total Low-Severity Issues	5	■ ■ ■ ■ ■
Total Informational-Severity Issues	11	■ ■ ■ ■ ■ ■ ■ ■ ■ ■ ■
Total Undetermined-Severity Issues	4	■ ■ ■ ■
Total	27	

Category Breakdown

Access Controls	1	■
Auditing and Logging	1	■
Configuration	6	■ ■ ■ ■ ■ ■
Data Validation	14	■ ■ ■ ■ ■ ■ ■ ■ ■ ■ ■ ■ ■ ■
Documentation	1	■
Patching	2	■ ■
Undefined Behavior	2	■ ■
Total	27	

Code Maturity Evaluation

Category Name	Description
Access Controls	**Satisfactory.** The project used a sufficient authentication and authorization system.
Arithmetic	**Satisfactory.** SafeMath was used throughout the system but was not tested with automated analysis tools. We identified a potential integer overflow, which could result in the return of incorrect values (TOB-FRAX-018).
Assembly Use/Low-Level Calls	**Weak.** The use of assembly was limited to necessary and common purposes such as executing `chainid` and `extcodesize`. However, we did find that the lack of an existence check on `delegatecall` could allow a call to fail silently (TOB-FRAX-011). The codebase also lacked return checks on many instances of the `transfer` and `transferFrom` functions (TOB-FRAX-001, TOB-FRAX-006, TOB-FRAX-007, TOB-FRAX-010, TOB-FRAX-025).
Centralization	**Moderate.** Throughout the system, it was easy to identify the privileged actors. These admin addresses (or governance) can update many critical protocol parameters. However, no user documentation on deployment or centralization risks was provided.
Code Stability	**Moderate.** Contracts were added to the codebase during the audit. We also identified discrepancies between deployed contracts and their counterparts in the repository.
Upgradeability	**Moderate.** `CurveAMO_V3` contained upgradeability mechanisms that used a proxy with a target implementation. The use of a `delegatecall` proxy pattern leaves the contract vulnerable to front-running attacks upon deployment (TOB-FRAX-009). Additionally, there was no continuous integration pipeline to which `slither-check-upgradeability` could be added.
Function Composition	**Satisfactory.** The code was reasonably well structured, and its functions had clear, narrow purposes. However, gas optimizations resulted in the duplication of certain logic.
Front-Running	**Moderate**. The use of the `delegatecall` proxy pattern leaves the `CurveAMO_V3` contract vulnerable to front-running attacks upon deployment (TOB-FRAX-009).
Monitoring	**Weak.** Many critical functions did not emit events, making it difficult to monitor the on-chain activity of the protocol. Additionally, we were not provided with an incident response plan or information on the use of off-chain components in behavior monitoring.

Specification	**Satisfactory.** Frax Finance provided comprehensive documentation, and the code contained an adequate number of comments.
Testing and Verification	**Moderate.** The system used unit tests but lacked more advanced testing techniques such as fuzzing or symbolic execution. Several critical functions were missing tests for exception paths, and only some of the tests were included in the public repository.

Engagement Goals

The engagement was scoped to provide a security assessment of the contracts in the `FraxFinance/frax-solidity` repository identified as high priority.

Specifically, we sought to answer the following questions:

- Are there appropriate access controls on the actions of users and admins?
- Is it possible for users to steal funds from others?
- Can participants manipulate the contracts in unexpected ways?
- Are there arithmetic overflows or underflows that affect the code?

Coverage

The engagement focused on the following components:

`FRAXStablecoin`. The FRAX token manages the pools, the global collateral ratio, and other important system parameters such as fees and oracles. We reviewed this component manually and through static analysis.

`FRAXShares`. The FXS token is used to track votes. We reviewed this component manually and through static analysis.

`veFXS`. This voting escrow contract is based on the one used by Curve. A user's voting power is proportional to the number of FXS shares that the user has locked up and the length of time during which they remain locked. We manually reviewed this contract.

`Pool_USDC` and the `FraxPool`. `Pool_USDC`, which inherits from the `FraxPool`, is a pool that holds a specific type of collateral and can be used to mint and redeem FRAX as well as to recollateralize and buy back FXS. We manually reviewed `Pool_USDC` and the `FraxPool` using static analysis and property-based fuzzing. Our property-based fuzzing efforts focused on ensuring that reverts and token transfers were present only where expected.

`CurveAMO_V3` implementation. This pool interacts with Curve and `FraxPool` to provide additional liquidity in USDC and FRAX. Users can accrue interest-bearing rewards on this additional liquidity (that is, excess `FraxPool` funds) and can mint CRV tokens to secure rewards. We reviewed this contract manually and by using our static analyzer Slither.

Recommendations Summary

This section aggregates all the recommendations made during the engagement. Short-term recommendations address the immediate causes of issues. Long-term recommendations pertain to the development process and long-term design goals.

Short Term

❑ **Always use `SafeERC20.safeTransfer` and `.safeTransferFrom` to ensure the proper handling of failed transfers, including token transfers that do not revert upon failing or have no return value.** TOB-FRAX-001, TOB-FRAX-006, TOB-FRAX-010, TOB-FRAX-025

❑ **Use a two-step process for all non-recoverable critical operations (such as `setNewOwner` and `acceptNewOwner`) to prevent irrevocable mistakes.** TOB-FRAX-002

❑ **Add events for all critical operations.** Events aid in contract monitoring and the detection of suspicious behavior. TOB-FRAX-003

❑ **Either rename the modifiers or standardize the rights that they grant.** TOB-FRAX-004

❑ **Add zero-value checks to the functions that lack them to ensure users cannot accidentally set incorrect values, misconfiguring the system.** TOB-FRAX-005

❑ **Document the risks associated with adding functionalities to `_beforeTokenTransfer` to ensure that future refactors will not introduce unexpected behavior.** TOB-FRAX-007

❑ **Measure the gas savings from optimizations and carefully weigh them against the possibility of an optimization-related bug.** TOB-FRAX-008

❑ **Use `hardhat-upgrades` to deploy the proxies and implementation contracts.** This will ensure that deployment scripts have robust protections against front-running attacks. TOB-FRAX-009

❑ **Implement a contract existence check before each `delegatecall`.** Document the fact that `suicide` and `selfdestruct` can lead to unexpected behavior, and prevent future upgrades from introducing these functions. TOB-FRAX-011

❑ **Consider improving Aragon's voting contract to mitigate the issues in the contract.** Alternatively, implement a different voting contract and perform a security assessment of the contract before deployment. TOB-FRAX-012

❑ **Model the risks that stem from the actions that whales can take on the platform, and consider establishing an upper bound on individual mints and redemptions to reduce those users' impact on the platform.** TOB-FRAX-013

❑ **Either remove the payable keyword from the `initialize()` function or document why a payable initialize function is necessary.** TOB-FRAX-014

❑ **Review the codebase and document the source and version of each dependency.** Include third-party sources as submodules in your Git repository to maintain internal path consistency and ensure that dependencies are updated periodically. TOB-FRAX-015

❑ **Develop documentation to inform users of what to do if a transaction fails because it has run out of gas.** TOB-FRAX-016

❑ **Review the implementations of `veFXS` and `CurveAMO_V3` and develop additional documentation on them.** TOB-FRAX-017

❑ **Either use SafeMath (that is, `usdc_subtotal.add(...)`) to cause a revert in the case of a failure or document the reason that native integer addition is used.** TOB-FRAX-018

❑ **Analyze the effects of a change in the global collateral ratio on the expected value of collateral.** TOB-FRAX-019

❑ **Clearly document for developers that `isContract()` is not guaranteed to return an accurate value, and emphasize that it should never be used to provide an assurance of security.** TOB-FRAX-020

❑ **Review all findings in the CurveDAO audit, identifying the risks associated with them and the mitigations that can be implemented to protect users.** TOB-FRAX-021

❑ **Ensure that `FraxPool` can handle tokens with more than 18 decimals.** TOB-FRAX-022

❑ **Ensure that scenarios in which the global collateral ratio exceeds the maximum are handled correctly and consistently, and check the behavior of the system when such scenarios arise.** Alternatively, ensure that the ratio can never exceed the maximum, such as by changing == to >= in `redeem1t1FRAX`. TOB-FRAX-023

❑ **Use neither `ABIEncoderV2` nor any other experimental Solidity feature.** Refactor the code such that structs do not need to be passed to or returned from functions. TOB-FRAX-024

❑ **Clearly communicate the differences between the public repository and the deployed contracts to auditors.** If possible, instruct auditors to work either solely from the contracts deployed on Etherscan or solely from the repository. TOB-FRAX-026

❑ **Establish reasonable upper bounds for fees and implement checks to ensure that the fees do not exceed them.** TOB-FRAX-027

Long Term

❑ **Integrate Slither into the continuous integration pipeline to catch missing return value checks, and review Appendix B, which outlines ERC20 token best practices.** TOB-FRAX-001, TOB-FRAX-006, TOB-FRAX-007, TOB-FRAX-010, TOB-FRAX-025

❑ **Identify and document all possible actions that can be taken by privileged accounts and their associated risks.** This will facilitate reviews of the codebase and prevent future mistakes. TOB-FRAX-002

❑ **Consider using a blockchain-monitoring system to track any suspicious behavior in the contracts.** The system relies on several contracts to behave as expected. A monitoring mechanism for critical events would quickly detect any compromised system components. TOB-FRAX-003

❑ **To prevent developers and users from making incorrect assumptions, use terms consistently within the components or throughout the system.** TOB-FRAX-004

❑ **Add checks to ensure that user-supplied arguments are not set to `address(0)`.** TOB-FRAX-005

❑ **Monitor the development and adoption of Solidity compiler optimizations to assess their maturity.** TOB-FRAX-008

❑ **Carefully review the Solidity documentation, especially the "Warnings" section, as well as the pitfalls of using the `delegatecall` proxy pattern.** TOB-FRAX-009, TOB-FRAX-011

❑ **Stay up to date on the latest research on blockchain-based online voting and bidding.** This research will continue to evolve, as there is not yet a perfect solution for the challenges of blockchain-based online voting. TOB-FRAX-012

❑ **Analyze all aspects of the system in which miners and whales can participate to understand the effects of their activities on the system.** TOB-FRAX-013

❑ **Use static analysis tools like [Slither](#) to detect structural issues such as contracts with non-retrievable funds.** TOB-FRAX-014

❑ **Use an Ethereum development environment and NPM to manage packages in the project.** TOB-FRAX-015

❑ **Investigate all loops used in the system to check whether they can run out of gas.** Focus on determining whether the number of iterations performed by a single loop can increase over time or can be influenced by users. TOB-FRAX-016

❑ **Consider writing an updated formal specification of the protocol.** TOB-FRAX-017

❑ **Use Echidna or Manticore to detect arithmetic overflows/underflows in the code.** TOB-FRAX-018

❑ **Analyze the implications of transaction atomicity for all blockchains in which this code will be deployed.** TOB-FRAX-019

❑ **Be mindful of the fact that the Ethereum core developers consider it poor practice to attempt to differentiate between end users and contracts.** Try to avoid this practice entirely if possible. TOB-FRAX-020

❑ **Always analyze the risk factors of integrations with third-party protocols and create an incident response plan prior to integration.** TOB-FRAX-021

❑ **Review the Token Integration Checklist and implement its recommendations on integrations with arbitrary tokens.** TOB-FRAX-022

❑ **Check that system values stay within the correct ranges and ensure that the system will not freeze if they exceed those ranges.** TOB-FRAX-023

❑ **Integrate static analysis tools like Slither into your CI pipeline to detect unsafe pragmas.** TOB-FRAX-024

❑ **Maintain consistency between deployed contracts and their files in the Git repository.** Each time a contract is deployed to the mainnet, “freeze” the file and its dependencies in GitHub. Instead of modifying the file of the deployed version, create a copy with a suffix (e.g., V2, V3, etc.) and work on that version until it is deployed to the mainnet. Then repeat the process. This will simplify future reviews and increase their precision. TOB-FRAX-026

❑ **Establish reasonable lower and upper bounds for all system parameters and implement a method of validating them.** This will prevent system participants from

accidentally or maliciously forcing the system into an unexpected or dysfunctional state.
TOB-FRAX-027

Findings Summary

#	Title	Type	Severity
1	Transfers of collateral tokens can silently fail, causing a loss of funds	Data Validation	High
2	Lack of two-step process for critical operations	Data Validation	High
3	Missing events for critical operations	Auditing and Logging	Informational
4	Inconsistent use of the term “governance”	Access Controls	Informational
5	Lack of zero check on functions	Data Validation	Informational
6	Lack of return value check in veFXS may result in failed ERC20 token recovery	Data Validation	Low
7	Lack of return value check in FXS may result in unexpected behavior	Data Validation	Informational
8	Solidity compiler optimizations can be problematic	Undefined Behavior	Informational
9	Initialization functions can be front-run	Configuration	High
10	Lack of return value check in CurveAMO_V3 may result in failed collateral retrieval	Data Validation	High
11	Lack of contract existence check on delegatecall will result in unexpected behavior	Data Validation	High
12	Aragon’s voting contract does not follow voting best practices	Configuration	High
13	Two-block delay may not deter whale activity	Configuration	Informational
14	Ether can be deposited into CurveAMO_V3 but not retrieved from it	Configuration	Low
15	Contracts used as dependencies do not track upstream changes	Patching	Low

16	External calls in loops may result in denial of service	Data Validation	Informational
17	Lack of contract and user documentation	Documentation	Informational
18	Use of Solidity arithmetic may result in integer overflows	Data Validation	Informational
19	Curve AMO assumes the collateral ratio to be constant	Data Validation	Undetermined
20	isContract() may behave unexpectedly	Undefined Behavior	Informational
21	Risks related to CurveDAO architecture	Configuration	High
22	Pool deployment will fail if collateral token has more than 18 decimals	Data Validation	Low
23	One-to-one minting and redeeming operations have different collateral ratio requirements	Data Validation	Undetermined
24	Use of non-production-ready ABIEncoder V2	Patching	Undetermined
25	Lack of return value check in Investor AMO contract	Data Validation	Low
26	Differences between public repository, deployed contracts, and private repository	Configuration	Informational
27	Owners and governance can set fees and other parameters to any value	Data Validation	Undetermined

1. Transfers of collateral tokens can silently fail, causing a loss of funds

Severity: High
Difficulty: Medium
Type: Data Validation
Finding ID: TOB-FRAX-001
Target: `FraxPool.sol`

Description
`FraxPool` does not check the return value of `collateral_token.transfer` or `collateral_token.transferFrom`. Certain tokens, such as BAT, return `false` rather than reverting if a transfer fails. If a pool for one of these tokens is added, users will be able to mint FRAX without providing collateral. This would break the system and the FRAX peg.

This issue is also present in the following functions:

- `FraxPool.mint1t1FRAX`
- `FraxPool.mintFractionalFRAX`
- `FraxPool.collectRedemption`
- `FraxPool.recollateralizeFRAX`
- `FraxPool.buyBackFXS`

Exploit Scenario
FRAX is collateralized by 80%. A new `FraxPool` with collateral token T is added to the system. The `transferFrom` function of this token returns `false` instead of reverting to signal a failure. Alice, who does not hold any T, calls `FraxPool.mintFractionalFRAX`. As long as Alice has enough FXS to mint the requested amount of FRAX, the transaction will succeed and Alice will have minted FRAX by paying only 20% of its value.

Recommendations
Short term, always use `SafeERC20.safeTransfer` and `.safeTransferFrom` to ensure the proper handling of failed transfers, including token transfers that do not revert upon failing or have no return value.

Long term, integrate Slither into the continuous integration pipeline to catch missing return value checks, and review Appendix B, which outlines ERC20 token best practices.

2. Lack of two-step process for critical operations

Severity: High
Difficulty: High
Type: Data Validation
Finding ID: TOB-FRAX-002
Target: `Frax.sol`, `FraxPool.sol`, `FXS.sol`

Description
Several critical operations are executed in one function call. This schema is error-prone and can lead to irrevocable mistakes.

For example, the `owner_address` variable defines the address that can add/remove pools, set parameters in the system, and update oracles. The setter function for this address immediately sets the new owner address:

```
function setOwner(address _owner_address) external onlyByOwnerOrGovernance {
    owner_address = _owner_address;
}
```

Figure 2.1: `contracts/Frax/Frax.sol#L259-L261`

If the address is incorrect, the protocol could permanently lose the ability to execute critical operations.

This issue is also present in the following contracts:
- `FraxPool.sol - setOwner`
- `FXS.sol - setOwner`

Exploit Scenario
Alice, a member of the Frax Finance team, sets a new address as the `owner`. However, because the new address includes a typo, the Frax Finance team loses the ability to add new pools. To address the issue, the team must deploy a new set of contracts with the correct owner.

Recommendations
Short term, use a two-step process for all non-recoverable critical operations (such as `setNewOwner` and `acceptNewOwner`) to prevent irrevocable mistakes.

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

3. Missing events for critical operations

Severity: Informational
Difficulty: Low
Type: Auditing and Logging
Finding ID: TOB-FRAX-003
Target: `Frax.sol, veFXS.vy, FXS.sol, FraxPool.sol, CurveAMO_V3.sol`

Description
Several critical operations do not trigger events. As a result, it is difficult to check the behavior of the contracts.

Ideally, the following critical operations should trigger events:

- `FRAXStablecoin`
  - `refreshCollateralRatio`
  - `addPool`
  - `removePool`
  - `setOwner`
  - `setRedemptionFee`
  - `setMintingFee`
  - `setFraxStep`
  - `setPriceTarget`
  - `setRefreshCooldown`
  - `setETHUSDOracle`
  - `setFXSAddress`
  - `setController`
  - `setPriceBand`
  - `setFRAXEthOracle`
  - `setFXSAddress`
- `veFXS`
  - `commit_smart_wallet_checker`
  - `apply_smart_wallet_checker`
  - `toggleEmergencyUnlock`
- `FRAXShares`
  - `setFRAXAddress`
  - `setFXSMinDAO`
  - `setOwner`
  - `toggleMinting`
  - `toggleRedeeming`
  - `toggleRecollateralize`
  - `toggleBuyBack`
  - `toggleCollateralPrice`
- `FraxPool`
  - `setPoolParameters`
  - `setTimelock`
  - `setOwner`
- `CurveAMO_V3`
  - `setTimelock`

- `setOwner`
- `setMiscRewardsCustodian`
- `setVoterContract`
- `setPool`
- `setThreePool`
- `setMetapool`
- `setVault`
- `setBorrowCap`
- `setMaxFraxOutstanding`
- `setMinimumCollateralRatio`
- `setConvergenceWindow`
- `setOverrideCollatBalance`
- `setCustomFloor`
- `setDiscountRate`
- `setSlippages`
- `recoverERC20`
- `mintRedeemPart1`
- `mintRedeemPart2`
- `burnFRAX`
- `burnFXS`
- `metapoolDeposit`

Without events, users and blockchain-monitoring systems cannot easily detect suspicious behavior.

Exploit Scenario
Eve compromises the `COLLATERAL_PRICE_PAUSER` role of `FraxPool`, calls `toggleCollateralPrice` with a very low price, and redeems her FRAX shares, draining a large amount of collateral from the pool. The Frax Finance team notices the change only when it is too late to mitigate it.

Recommendations
Short term, add events for all critical operations. Events aid in contract monitoring and the detection of suspicious behavior.

Long term, consider using a blockchain-monitoring system to track any suspicious behavior in the contracts. The system relies on several contracts to behave as expected. A monitoring mechanism for critical events would quickly detect any compromised system components.

4. Inconsistent use of the term “governance”

Severity: Informational
Type: Access Controls
Target: `Frax.sol`

Difficulty: High
Finding ID: TOB-FRAX-004

Description
`FRAXStablecoin` contains the modifiers `onlyByOwnerOrGovernance` and `onlyByOwnerGovernanceOrPool`, which use the word “governance” in different ways. In the former, it refers to the `timelock_address` or `controller_address`, and in the latter, it refers only to the `timelock_address`. This inconsistency may cause the introduction of errors during development.

```
 modifier onlyByOwnerOrGovernance() {
     require(msg.sender == owner_address || msg.sender == timelock_address ||
msg.sender == controller_address, "You are not the owner, controller, or the
governance timelock");
     _;
 }

 modifier onlyByOwnerGovernanceOrPool() {
     require(
         msg.sender == owner_address
         || msg.sender == timelock_address
         || frax_pools[msg.sender] == true,
         "You are not the owner, the governance timelock, or a pool");
     _;
  }
```

Figure 4.1: `contracts/Frax/Frax.sol#L92-L104`

Exploit Scenario
Developer Bob adds the modifier `onlyByOwnerOrGovernance`, thinking it is callable only by the `owner_address` and `timelock_address`. However, it is also callable by the `controller_address`, meaning that this party can unexpectedly execute privileged operations.

Recommendations
Short term, either rename the modifiers or standardize the rights that they grant.

Long term, to prevent developers and users from making incorrect assumptions, use terms consistently within the components or throughout the system.

5. Lack of zero check on functions

Severity: Informational
Type: Data Validation
Target: `contracts/*`

Difficulty: High
Finding ID: TOB-FRAX-005

Description
Certain setter functions fail to validate incoming arguments, so callers can accidentally set important state variables to the zero address.

For example, `FXS`' `setOwner` function sets the owner address meant to interact with tokens to calculate values:

```
function setOwner(address _owner_address) external onlyByOwnerOrGovernance {
    owner_address = _owner_address;
}
```

Figure 5.1: `FXS/FXS.sol#L115-L117`

Immediately after this address has been set to `address(0)`, the admin must reset the value; failure to do so may result in unexpected contract behavior.

This issue is also present in the following contracts:

- `FraxPool.sol`
  - `constructor - _frax_contract_address, _fxs_contract_address, _collateral_address, _creator_address, _timelock_address`
  - `setCollatETHOracle - _collateral_weth_oracle_address, _weth_address`
  - `setTimelock - new_timelock`
  - `setOwner - _owner_address`
- `Frax.sol`
  - `constructor - _creator_address, _timelock_address`
  - `setOwner - _owner_address`
  - `setFXSAddress - _fxsAddress`
  - `setETHUSDOracle - _eth_usd_consumer_address`
  - `setTimelock - new_timelock`
  - `setController - _controller_address`
  - `setPriceBand - _price_band`
  - `setFXSEthOracle - _fxs_oracle_addr, _weth_address`
  - `setFRAXEthOracle - _frax_oracle_addr, _weth_address`
- `FXS.sol`
  - `constructor - _owner_address, _oracle_address, _timelock_address`
  - `setOwner - _owner_address`
  - `setOracle - new_oracle`

    - `setTimelock - new_timelock`
    - `setFRAXAddress - frax_contract_address`
- `Pool_USDC.sol`
    - `constructor - _frax_contract_address, _fxs_contract_address, _collateral_address, _creator_address, _timelock_address, _pool_ceiling`

Exploit Scenario
Alice sets up a multisig that she wants to set as the new address. When she invokes `setOwner` to replace the address, she accidentally enters the zero address. As a result, only a governance process will be able to reset the address, if it can be reset at all.

Recommendations
Short term, add zero-value checks to the functions mentioned above to ensure users cannot accidentally set incorrect values, misconfiguring the system.

Long term, add checks to ensure that user-supplied arguments are not set to `address(0)`.

6. Lack of return value check in veFXS may result in failed ERC20 token recovery

Severity: Low
Type: Data Validation
Target: `contracts/Curve/veFXS.vy, CurveAMO_V3.sol`

Difficulty: High
Finding ID: TOB-FRAX-006

Description
The `veFXS` contract does not check the return value of a call to transfer tokens from the `veFXS` contract to the admin. Without this check, such transfers could fail.

```
@external
def recoverERC20(token_addr: address, amount: uint256):
    """
    @dev Used to recover non-FXS ERC20 tokens
    """
    assert msg.sender == self.admin  # dev: admin only
    assert token_addr != self.token  # Cannot recover FXS. Use toggleEmergencyUnlock instead
and have users pull theirs out individually
    ERC20(token_addr).transfer(self.admin, amount)
```

Figure 6.1: `contracts/Curve/veFXS.vy#L222-L229`

If the target token implementation returns `false` instead of reverting, the `recoverERC20` function may not detect the failed transfer call. Instead, the `recoverERC20` function may return `true` despite its failure to transfer tokens to the admin.

This issue is also present in `CurveAMO_V3`'s `recoverERC20` function.

Exploit Scenario
Alice, the owner of a contract, calls the `recoverERC20` function to rescue a user's funds. The contract interacts with a token that returns `false` instead of reverting, such as BAT. Because of a lack of funds, the token transfer call fails. When Alice invokes the contract, the tokens are not actually sent, but the transaction succeeds.

Recommendations
Short term, either wrap the `transfer` call in a `require` statement or use a `safeTransfer` function. Taking either step will ensure that if a transfer fails, the transaction will also fail.

Long term, integrate [Slither](https://github.com/crytic/slither) into the continuous integration pipeline to catch missing return value checks.

7. Lack of return value check in FXS may result in unexpected behavior

Severity: Informational
Type: Data Validation
Target: `FraxPool.sol`

Difficulty: High
Finding ID: TOB-FRAX-007

Description
The `FraxPool` contract does not check the return value of a call to transfer FXS tokens. Without this check, the FRAX system may exhibit unexpected behavior.

The `collectRedemption` function calls FRAX's `ERC20` transfer function to transfer tokens from the pool to a user's account:

```
    // After a redemption happens, transfer the newly minted FXS and owed
collateral from this pool
    // contract to the user. Redemption is split into two functions to prevent
flash loans from being able
    // to take out FRAX/collateral from the system, use an AMM to trade the new
price, and then mint back into the system.
    function collectRedemption() external {
        require((lastRedeemed[msg.sender].add(redemption_delay)) <= block.number,
"Must wait for redemption_delay blocks before collecting redemption");
        bool sendFXS = false;
        bool sendCollateral = false;
        uint FXSAmount;
        uint CollateralAmount;

        [...]

        if(sendFXS == true){
            FXS.transfer(msg.sender, FXSAmount);
        }
        if(sendCollateral == true){
            collateral_token.transfer(msg.sender, CollateralAmount);
        }
    }
```

Figure 7.1: `contracts/Frax/Pools/FraxPool.sol#L336-L369`

This transfer functionality is implemented in the `ERC20Custom` token contract:

```
    function _transfer(address sender, address recipient, uint256 amount) internal
virtual {
        require(sender != address(0), "ERC20: transfer from the zero address");
        require(recipient != address(0), "ERC20: transfer to the zero address");

        _beforeTokenTransfer(sender, recipient, amount);

        _balances[sender] = _balances[sender].sub(amount, "ERC20: transfer amount
exceeds balance");
        _balances[recipient] = _balances[recipient].add(amount);
        emit Transfer(sender, recipient, amount);
    }
```

Figure 7.2: `contracts/ERC20/ERC20Custom.sol#L159-L168`

The `_beforeTokenTransfer` hook is not currently used. If this function is used in the future, Frax Finance should ensure it reverts upon a failure.

Exploit Scenario
Alice, a member of the Frax Finance team, adds a new functionality to the `_beforeTokenTransfer` hook that returns `false` upon failing. Neither the `_transfer()` function nor the pool checks for calls that return `false` when they fail. As a result, the failure is ignored, and the transfer still occurs.

Recommendations
Short term, document the risks associated with adding functionalities to `_beforeTokenTransfer` to ensure that future refactors will not introduce unexpected behavior.

Long term, integrate [Slither](https://github.com/crytic/slither) into the continuous integration pipeline to catch missing return value checks.

8. Solidity compiler optimizations can be problematic

Severity: Informational
Type: Undefined Behavior
Target: `truffle-config.js`

Difficulty: Low
Finding ID: TOB-FRAX-008

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—or in the Emscripten transpilation to `solc-js`—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.

9. Initialization functions can be front-run

Severity: High
Difficulty: High
Type: Configuration
Finding ID: TOB-FRAX-009
Target: `CurveAMO_V3.sol`

Description
Several implementation contracts have initialization functions that can be front-run, allowing an attacker to incorrectly initialize the contracts.

Due to the use of the `delegatecall` proxy pattern, many of the contracts cannot be initialized with a constructor and have initializer functions:

```
function initialize(
    address _frax_contract_address,
    address _fxs_contract_address,
    address _collateral_address,
    address _creator_address,
    address _custodian_address,
    address _timelock_address,
    address _frax3crv_metapool_address,
    address _three_pool_address,
    address _three_pool_token_address,
    address _pool_address
) public payable initializer {
    FRAX = FRAXStablecoin(_frax_contract_address);
    fxs_contract_address = _fxs_contract_address;
    collateral_token_address = _collateral_address;
    collateral_token = ERC20(_collateral_address);
    crv_address = 0xD533a949740bb3306d119CC777fa900bA034cd52;
    missing_decimals = uint(18).sub(collateral_token.decimals());
    timelock_address = _timelock_address;
    owner_address = _creator_address;
    custodian_address = _custodian_address;
    voter_contract_address = _custodian_address; // Default to the custodian
}
```

Figure 9.1: `contracts/Curve/CurveAMO_V3.sol#L109-L130`

An attacker could front-run these functions and initialize the contracts with malicious values.

Exploit Scenario
Bob deploys the `CurveAMO_V3` contract. Eve front-runs the contract initialization and sets her own address as the `_collateral_address` value. As a result, she can set tokens that return booleans and do not revert upon failing, thereby exploiting the lack of return value checks (TOB-FRAX-001).

Recommendations
Short term, use `hardhat-upgrades` to deploy the proxies and implementation contracts. This will ensure that deployment scripts have robust protections against front-running attacks.

Long term, carefully review the Solidity documentation, especially the "Warnings" section, as well as the pitfalls of using the `delegatecall` proxy pattern.

10. Lack of return value check in CurveAMO_V3 may result in failed collateral retrieval

Severity: High
Difficulty: High
Type: Data Validation
Finding ID: TOB-FRAX-010
Target: `CurveAMO_V3.sol`

Description

The `CurveAMO_V3` contract does not check the return value of a call to transfer tokens from the `collateral_token` contract to the pool address. Without this check, such transfers could fail.

```
// Give USDC profits back
function giveCollatBack(uint256 amount) external onlyByOwnerOrGovernance {
    collateral_token.transfer(address(pool), amount);
    returned_collat_historical = returned_collat_historical.add(amount);
}
```

Figure 10.1: `contracts/Curve/CurveAMO_V3.sol#L339-L343`

If the target token implementation returns `false` instead of reverting, the `giveCollatBack` function may not detect the failed transfer call. Instead, the `giveCollatBack` function may return `true` despite its failure to transfer tokens to the pool.

This issue is also present in `CurveAMO_V3.withdrawCRVRewards`.

Exploit Scenario

Alice, the owner of a contract, calls the `giveCollatBack` function to retrieve USDC profits. The contract interacts with a token that returns `false` instead of reverting, such as BAT. Because of a lack of funds, the token transfer call fails. When Alice invokes the contract, the tokens are not actually sent, but the transaction succeeds.

Recommendations

Short term, either wrap the `transfer` call in a `require` statement or use a `safeTransfer` function. Taking either step will ensure that if a transfer fails, the transaction will also fail.

Long term, integrate [Slither](https://github.com/crytic/slither) into the continuous integration pipeline to catch missing return value checks.

11. Lack of contract existence check on delegatecall will result in unexpected behavior

Severity: High
Difficulty: High
Type: Data Validation
Finding ID: TOB-FRAX-011
Target: `import.sol`

Description

The `AdminUpgradeableProxy` contract uses the `delegatecall` proxy pattern. If the implementation contract is incorrectly set or is self-destructed, the proxy may not detect failed executions.

The `AdminUpgradeableProxy` contract uses `ERC1967Proxy`, which inherits from a chain of OpenZeppelin contracts such as `ERC1967Upgrade` and `Proxy`. Eventually, arbitrary calls are executed by the `_fallback` function in the `Proxy`, which lacks a contract existence check:

```
    function _delegate(address implementation) internal virtual {
        // solhint-disable-next-line no-inline-assembly
        assembly {
            // Copy msg.data. We take full control of memory in this inline assembly
            // block because it will not return to Solidity code. We overwrite the
            // Solidity scratch pad at memory position 0.
            calldatacopy(0, 0, calldatasize())

            // Call the implementation.
            // out and outsize are 0 because we don't know the size yet.
            let result := delegatecall(gas(), implementation, 0, calldatasize(), 0, 0)

            // Copy the returned data.
            returndatacopy(0, 0, returndatasize())

            switch result
            // delegatecall returns 0 on error.
            case 0 { revert(0, returndatasize()) }
            default { return(0, returndatasize()) }
        }
    }

    /**
     * @dev This is a virtual function that should be overriden so it returns the address to
which the fallback function
```

```
     * and {_fallback} should delegate.
     */
    function _implementation() internal view virtual returns (address);

    /**
     * @dev Delegates the current call to the address returned by `_implementation()`.
     *
     * This function does not return to its internall call site, it will return directly to
the external caller.
     */
    function _fallback() internal virtual {
        _beforeFallback();
        _delegate(_implementation());
    }
```

Figure 11.1: *`Proxy.sol#L21-L57`*

As a result, a `delegatecall` to a destructed contract will return success as part of the EVM specification. The [Solidity documentation](#) includes the following warning:

> *The low-level call, delegatecall and callcode will return success if the called account is non-existent, as part of the design of EVM. Existence must be checked prior to calling if desired.*

Figure 11.2: A snippet of the Solidity documentation detailing unexpected behavior related to *`delegatecall`*

The proxy will not throw an error if its implementation is incorrectly set or self-destructed. It will instead return success even though no code was executed.

Exploit Scenario
Eve upgrades the proxy to point to an implementation address that is not the address of a current contract. As a result, each `delegatecall` returns success without changing the state or executing code. Eve uses this failing to scam users.

Recommendations
Short term, implement a contract existence check before each `delegatecall`. Document the fact that `suicide` and `selfdestruct` can lead to unexpected behavior, and prevent future upgrades from introducing these functions.

Long term, carefully review the Solidity documentation, especially the "Warnings" section, as well as the pitfalls of using the `delegatecall` proxy pattern.

References
- Contract Upgrade Anti-Patterns

12. Aragon's voting contract does not follow voting best practices

Severity: High
Difficulty: Medium
Type: Configuration
Finding ID: TOB-FRAX-012
Target: Aragon's `Voting.sol`

Description
`veFXS` uses the Aragon contract for voting. While its voting logic is simple, it fails to prevent several potential abuses of on-chain voting processes.

The voting contract has the following issues:

- It lacks a way to mitigate the use of a "quick vote and withdraw" strategy.
- It does not provide incentives for early voting.
- It has no mitigations for spam attacks. An attacker with vote-creation rights could create hundreds of thousands of votes and would need only one to pass to succeed.
- It uses a significantly outdated Solidity version.
- It has not had any updates released since July 2020.

Exploit Scenario
Eve, a miner, creates new votes to set a new `minter` on `ERC20CRV` on every block. Other users cannot participate in all of the votes. As a result, one vote is accepted, and Eve takes control of `ERC20CRV`'s minting.

Recommendations
Short term, consider improving Aragon's voting contract to mitigate the above issues. Alternatively, implement a different voting contract and perform a security assessment of the contract before deployment.

Long term, stay up to date on the latest research on blockchain-based online voting and bidding. This research will continue to evolve, as there is not yet a perfect solution for the challenges of blockchain-based online voting.

References
- Vocdoni
- Security Disclosure: Aragon 0.6 Voting ("Voting v1")
- Aragon vote shows the perils of on-chain governance

13. Two-block delay may not deter whale activity

Severity: Informational
Difficulty: High
Type: Configuration
Finding ID: TOB-FRAX-013
Target: `FraxPool.sol`

Description
Frax Finance has purposely implemented a two-block delay between a redemption request and the disbursement of that redemption. The delay is intended to prevent participants from conducting riskless arbitrage using flash loans.

However, whales, users who have a significant amount of funds on hand, can use their funds in the same way that flash loans are used. As a result, Frax Finance will need to consider the implications of whales on the system.

Additionally, miners may be able to silently push redemption requests and subsequently publish them to the blockchain. Frax Finance should analyze the risks that these privileged blockchain users pose to the system.

Exploit Scenario
Charlie, a whale who owns a lot of Ether, uses his funds to mint FRAX and then to repeatedly redeem his FRAX shares to engage in arbitrage. Because of the difference in the tokens' collateral ratios, he is able to realize a larger USDC profit than he is entitled to.

Recommendations
Short term, model the risks that stem from the actions that whales can take on the platform, and consider establishing an upper bound on individual mints and redemptions to reduce those users' impact on the platform.

Long term, analyze all aspects of the system in which miners and whales can participate to understand the effects of their activities on the system.

14. Ether can be deposited into CurveAMO_V3 but not retrieved from it

Severity: Low
Type: Configuration
Target: `CurveAMO_V3.sol`

Difficulty: Medium
Finding ID: TOB-FRAX-014

Description
The `CurveAMO_V3` contract has a `payable initialize()` function but lacks a function for withdrawing the funds. This issue is somewhat mitigated, however, by the fact that the Curve AMO contract operates by delegating withdrawals to a proxy capable of extracting funds.

```
function initialize(
        address _frax_contract_address,
        address _fxs_contract_address,
        address _collateral_address,
        address _creator_address,
        address _custodian_address,
        address _timelock_address,
        address _frax3crv_metapool_address,
        address _three_pool_address,
        address _three_pool_token_address,
        address _pool_address
    ) public payable initializer {
```

Figure 14.1: `contracts/Curve/CurveAMO_V3.sol#L109-L120`

Exploit Scenario
Alice, a member of the Frax Finance team, calls `initialize()` with ETH and is subsequently unable to directly retrieve the transferred funds from the contract.

Recommendations
Short term, either remove the payable keyword from the `initialize()` function or document why a payable initialize function is necessary.

Long term, use static analysis tools like [Slither](https://github.com/crytic/slither) to detect structural issues such as contracts with non-retrievable funds.

15. Contracts used as dependencies do not track upstream changes

Severity: Low
Difficulty: Low
Type: Patching
Finding ID: TOB-FRAX-015
Target: `contracts/library/*`

Description
Several third-party contracts have been copied and pasted into the Frax Finance repository, including into files such as `Address`, `ERC20`, `Babylonian`, `Governance`, and `Uniswap interfaces`. The code documentation does not specify the exact revision that was made or whether the code was modified. As such, the contracts will not reliably reflect updates or security fixes implemented in their dependencies, as those changes must be manually integrated into the contracts.

Exploit Scenario
A third-party contract used in FRAX receives an update with a critical fix for a vulnerability. An attacker detects the use of a vulnerable contract and can then exploit the vulnerability against any of the contracts in the `library`.

Recommendations
Short term, review the codebase and document the source and version of each dependency. Include third-party sources as submodules in your Git repository to maintain internal path consistency and ensure that dependencies are updated periodically.

Long term, use an Ethereum development environment and NPM to manage packages in the project.

16. External calls in loops may result in denial of service

Severity: Informational	Difficulty: Medium
Type: Data Validation	Finding ID: TOB-FRAX-016

Target: `contracts/Curve/veFXS.vy`, `contracts/Frax/Frax.sol`, `CurveAMO_V3.sol`

Description
The use of external calls in nested loops and subsequent loops, which iterate over lists that could have been provided by callers, may result in an out-of-gas failure during execution.

To determine the total value of the collateral in the FRAX system, the code loops over all `frax_pools` to retrieve the collateral balance (in dollars):

```
 // Iterate through all frax pools and calculate all value of collateral in all pools
globally
 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 16.1: `contracts/Curve/CurveAMO_V3.sol#L256-L265`

This issue is also present in the following functions:

- `CurveAMO_V3.iterate()`
- `veFXS._checkpoint()`

Exploit Scenario
Alice, a user, tries to retrieve the total balance of the collateral in the FRAX system. The execution runs out of gas during the computation, and an administrator must remove pools before the total collateral value can be determined.

Recommendations
Short term, develop documentation to inform users of what to do if a transaction fails because it has run out of gas.

Long term, investigate all loops used in the system to check whether they can run out of gas. Focus on determining whether the number of iterations performed by a single loop can increase over time or can be influenced by users.

17. Lack of contract and user documentation

Severity: Informational
Difficulty: Low
Type: Documentation
Finding ID: TOB-FRAX-017
Target: `veFXS.vy`, `CurveAMO_V3.sol`

Description
Parts of the codebase lack code documentation, high-level descriptions, and examples, making the contracts difficult to review and increasing the likelihood of user mistakes.

The documentation would benefit from the following details:

- `veFXS`
  - An explanation of how the `smart_wallet_checker` determines what to include on its whitelist
  - An analysis of the process of calculating a vote's weight (for the benefit of users and developers)
  - User documentation regarding the rounding down of the lock time by a week
- `CurveAMO_V3`
  - Clear user documentation regarding scenarios in which the value of a user's deposit may be lower than expected or the user may be able to withdraw more funds than expected
  - Details on the circumstances in which a custom floor and discount rate would be set
  - Additional user documentation regarding the return value of `showAllocations()`

The documentation on each of these items should include its expected properties and assumptions.

Recommendations
Short term, review and properly document the items mentioned above.

Long term, consider writing an updated formal specification of the protocol.

18. Use of Solidity arithmetic may result in integer overflows

Severity: Informational
Type: Data Validation
Target: `veFXS.vy`, `CurveAMO_V3.sol`

Difficulty: Low
Finding ID: TOB-FRAX-018

Description

The `showAllocations()` function returns an array of values, one of which is calculated from the sum of `usdc_subtotal` (which is a `uint256`) and another integer expression. The two values are added using the + operator, which performs native Solidity integer addition without checking for overflows. As such, if the sum is too large a value, it will cause a wraparound, which could lead to unexpected behavior.

```
        return [
            frax_in_contract, // [0]
            frax_withdrawable, // [1]
            frax_withdrawable.add(frax_in_contract), // [2]
            usdc_in_contract, // [3]
            usdc_withdrawable, // [4]
            usdc_subtotal, // [5]
            usdc_subtotal +
(frax_in_contract.add(frax_withdrawable)).mul(fraxDiscountRate()).div(1e6 * (10 **
missing_decimals)), // [6] USDC Total
            lp_owned, // [7]
            frax3crv_supply, // [8]
            _3pool_withdrawable, // [9]
            lp_value_in_vault // [10]
        ];
```

Figure 18.1: `contracts/Curve/CurveAMO_V3.sol#L223-L235`

Recommendations

Short term, either use `SafeMath` (that is, `usdc_subtotal.add(...)`) to cause a revert in the case of a failure or document the reason that native integer addition is used.

Long term, use [Echidna](https://github.com/crytic/echidna) or [Manticore](https://github.com/trailofbits/manticore) to detect arithmetic overflows/underflows in the code.

19. Curve AMO assumes the collateral ratio to be constant

Severity: Undetermined
Type: Data Validation
Target: `CurveAMO_V3.sol`

Difficulty: Medium
Finding ID: TOB-FRAX-019

Description
When calculating the amount of collateral available to the AMO, the `CurveAMO_V3` contract assumes that the collateral ratio will not change:

```
The protocol calculates the amount of underlying collateral the AMO has access to by finding the balance of USDC it can withdraw if the price of FRAX were to drop to the CR. Since FRAX is always backed by collateral at the value of the CR, it should never go below the value of the collateral itself. For example, FRAX should never go below $.85 at an 85% CR. This calculation is the safest and most conservative way to calculate the amount of collateral the Curve AMO has access to. This allows the Curve AMO to mint FRAX to place inside the pool in addition to USDC collateral to tighten the peg while knowing exactly how much collateral it has access to if FRAX were to break its peg.
```

Figure 19.1: [Frax Curve Documentation]()

This assumption is evident in the `mintRedeemPart1` function, which is used to retrieve the global collateral ratio of the stablecoin protocol:

```
    // This is basically a workaround to transfer USDC from the FraxPool to this investor contract
    // This contract is essentially marked as a 'pool' so it can call OnlyPools functions like pool_mint and pool_burn_from
    // on the main FRAX contract
    // It mints FRAX from nothing, and redeems it on the target pool for collateral and FXS
    // The burn can be called separately later on
    function mintRedeemPart1(uint256 frax_amount) external onlyByOwnerOrGovernance {
        //require(allow_yearn || allow_aave || allow_compound, 'All strategies are currently off');
        uint256 redemption_fee = pool.redemption_fee();
        uint256 col_price_usd = pool.getCollateralPrice();
        uint256 global_collateral_ratio = FRAX.global_collateral_ratio();
        uint256 redeem_amount_E6 = (frax_amount.mul(uint256(1e6).sub(redemption_fee))).div(1e6).div(10 ** missing_decimals);
        uint256 expected_collat_amount = redeem_amount_E6.mul(global_collateral_ratio).div(1e6);
```

```
        expected_collat_amount = expected_collat_amount.mul(1e6).div(col_price_usd);
```

Figure 19.2: `contracts/Curve/CurveAMO_V3.sol#L310-L322`

The contract uses the following formula to determine the value of `expected_collat_amount`:

$$expected\ collat\ amount = \frac{frax\ amount \cdot 1e6 - redemption\ fee}{1e6 \cdot missing\ decimals} \cdot \frac{global\ collateral\ ratio}{1e6} \cdot \frac{1e6}{collateral\ price\ in\ USD}$$
$$expected\ collat\ amount = \frac{frax\ amount \cdot 1e6 - redemption\ fee}{1e6 \cdot missing\ decimals} \cdot \frac{global\ collateral\ ratio}{collateral\ price\ in\ USD}$$

Curve AMO uses the pool to establish a price floor for FRAX. However, if the global collateral ratio changes (violating the contract's assumption that it will not), the available collateral will scale up or down, depending on the change.

Exploit Scenario
Alice, a Frax Finance administrator, calls `mintRedeemPart1` to transfer USDC from the `FraxPool` to the `CurveAMO_V3` contract. However, the global collateral ratio simultaneously decreases, so the contract receives less collateral than expected.

Recommendations
Short term, analyze the effects of a change in the global collateral ratio on the expected value of collateral.

Long term, analyze the implications of transaction atomicity for all blockchains in which this code will be deployed.

20. isContract() may behave unexpectedly

Severity: Informational
Difficulty: N/A
Type: Undefined Behavior
Finding ID: TOB-FRAX-020
Target: `FRAXStablecoin/Address.sol`

Description
The FRAX system relies on the `isContract()` function in `Address.sol` to check whether there is a contract at the target address. However, in Solidity, there is no general way to definitively determine that, as there are several edge cases in which the underlying function `extcodesize()` can return unexpected results. In addition, there is no way to guarantee that an address that *is* that of a contract (or one that is not) will remain that way in the future.

```
function isContract(address account) internal view returns (bool) {
    // This method relies in extcodesize, which returns 0 for contracts in
    // construction, since the code is only stored at the end of the
    // constructor execution.

    uint256 size;
    // solhint-disable-next-line no-inline-assembly
    assembly { size := extcodesize(account) }
    return size > 0;
}
```

Figure 20.1: `FRAXStablecoin/Address.sol#L25-L34`

Exploit Scenario
A function, *f*, within the FRAX codebase calls `isContract()` internally to guarantee that a certain method is not callable by another contract. An attacker creates a contract that calls *f* from within its constructor, and the call to `isContract()` within *f* returns `false`, violating the “guarantee.”

Recommendations
Short term, clearly document for developers that `isContract()` is not guaranteed to return an accurate value, and emphasize that it should never be used to provide an assurance of security.

Long term, be mindful of the fact that the Ethereum core developers consider it poor practice to attempt to differentiate between end users and contracts. Try to avoid this practice entirely if possible.

21. Risks related to CurveDAO architecture

Severity: High
Type: Configuration
Target: `CurveAMO_V3.sol`

Difficulty: Medium
Finding ID: TOB-FRAX-021

Description
`CurveAMO_V3` relies heavily on `Curve Pools` to tighten the stable FRAX peg and to distribute CRV to token holders, granting them voting rights in CurveDAO.

The use of CurveDAO could affect the FRAX protocol in a few ways. Frax Finance should be mindful of the following considerations:

- If the `kick` function in `LiquidityGauge` is not monitored, users who abuse the system will not be penalized. (See TOB-CURVE-DAO-001.)
- It will be necessary to ensure that rewards are distributed to users fairly.
- The differences between calls to `balanceOfAt` and `totalSupplyAt` should be documented. (See TOB-CURVE-DAO-016.)

Exploit Scenario
Alice, a FRAX user, interacts with code that has been part of the CurveDAO architecture since inception. Because of existing vulnerabilities that favor early users of CurveDAO, she receives a higher amount of CRV rewards than she is entitled to.

Recommendations
Short term, review all findings in the CurveDAO audit, identifying the risks associated with them and the mitigations that can be implemented to protect users.

Long term, always analyze the risk factors of integrations with third-party protocols and create an incident response plan prior to integration.

References
- CurveDAO Audit

22. Pool deployment will fail if collateral token has more than 18 decimals

Severity: Low
Difficulty: Low
Type: Data Validation
Finding ID: TOB-FRAX-022
Target: `FraxPool.sol`

Description

To account for tokens with decimal values other than 18, `FraxPool` computes the `missing_decimals` and then scales the token amount by `10**18`. This process succeeds for tokens with fewer than 18 decimals but fails for those with more than 18.

```
missing_decimals = uint(18).sub(collateral_token.decimals());
```

Figure 22.1: `contracts/Frax/Pools/FraxPool.sol#120`

Exploit Scenario

Governance wants to deploy a new pool for a collateral token with 20 decimals. The deployment fails, making it impossible to use that type of collateral.

Recommendations

Short term, ensure that `FraxPool` can handle tokens with more than 18 decimals.

Long term, review the [Token Integration Checklist] and implement its recommendations on integrations with arbitrary tokens.

23. One-to-one minting and redeeming operations have different collateral ratio requirements

Severity: Undetermined
Type: Data Validation
Target: `FraxPool.sol`

Difficulty: Low
Finding ID: TOB-FRAX-023

Description
`mint1t1FRAX` checks that the global collateral ratio is greater than or equal to the maximum global collateral ratio:

```
require(FRAX.global_collateral_ratio() >= COLLATERAL_RATIO_MAX, "Collateral ratio must be >=
1");
```

Figure 23.1: `contracts/Frax/Pools/FraxPool.sol#179`

By contrast, `redeem1t1FRAX` checks that the global collateral ratio is equal to the maximum global collateral ratio:

```
require(FRAX.global_collateral_ratio() == COLLATERAL_RATIO_MAX, "Collateral ratio must be ==
1");
```

Figure 23.2: `contracts/Frax/Pools/FraxPool.sol#242`

If the collateral ratio ever exceeds the maximum, minting will still be possible but redeeming will not be.

Exploit Scenario
The global collateral ratio exceeds the maximum global collateral ratio. Alice, a user, can mint FRAX but cannot redeem it until the global collateral ratio decreases to the maximum ratio.

Recommendations
Short term, ensure that scenarios in which the global collateral ratio exceeds the maximum are handled correctly and consistently, and check the behavior of the system when such scenarios arise. Alternatively, ensure that the ratio can never exceed the maximum, such as by changing `==` to `>=` in `redeem1t1FRAX`.

Long term, check that system values stay within the correct ranges and ensure that the system will not freeze if they exceed those ranges.

24. Use of non-production-ready ABIEncoderV2

Severity: Undetermined
Type: Patching
Target: `Throughout`

Difficulty: Low
Finding ID: TOB-FRAX-024

Description
The contracts use the new Solidity ABI encoder, `ABIEncoderV2`. This experimental encoder is not ready for production.

More than 3% of all GitHub issues for the Solidity compiler are related to experimental features, primarily `ABIEncoderV2`. Several issues and bug reports are still open and unresolved. `ABIEncoderV2` has been associated with more than 20 high-severity bugs, some of which are so recent that fixes for them have not yet been included in a Solidity release.

For example, in March 2019, a severe bug introduced in Solidity 0.5.5 was found in the encoder.

The following contracts use this encoder:

- `Frax.sol`
- `FraxPool.sol`
- `FraxPoolLibrary.sol`
- `FXS.sol`

Exploit Scenario
The `CurveAMO_V3` contract is deployed. After deployment, a bug is found in the encoder, which means that the contract is broken and can be exploited.

Recommendations
Short term, use neither `ABIEncoderV2` nor any other experimental Solidity feature. Refactor the code such that structs do not need to be passed to or returned from functions.

Long term, integrate static analysis tools like Slither into your CI pipeline to detect unsafe pragmas.

25. Lack of return value check in Investor AMO contract

Severity: Low
Type: Data Validation
Target: `contracts/Misc_AMOs/InvestorAMO_V2.sol`

Difficulty: High
Finding ID: TOB-FRAX-025

Description

The function `giveCollatBack()` is used in the Investor AMO contract to repay collateral to a pool. However, there is no check on the return value of the final `transfer()` call, meaning that if the repayment operation fails, the contract's `borrowed_balance` will still decrease.

```
function giveCollatBack(uint256 amount) public onlyByOwnerOrGovernance {
    // Still paying back principal
    if (amount <= borrowed_balance) {
        borrowed_balance = borrowed_balance.sub(amount);
    }
    // Pure profits
    else {
        borrowed_balance = 0;
    }
    paid_back_historical = paid_back_historical.add(amount);
    collateral_token.transfer(address(pool), amount);
}
```

Figure 25.1: `InvestorAMO_V2.sol#L252-L263`

The `withdrawRewards()` and `emergencyRecoverERC20()` functions in the AMO contract code also fail to check the return value of `transfer()` calls. However, because neither modifies the contract's state, the lack of a check is less problematic.

Exploit Scenario

An attacker with owner or governance rights finds a way to cause transfers to fail when collateral tokens are returned to the pool and then calls `giveCollatBack()`. `InvestorAMO_V2` records the transaction, indicating that part of its balance was paid off when in fact the transfer failed and the tokens remain in the contract.

Recommendations

Short term, either wrap the `transfer` call in a `require` statement or use a `safeTransfer` function. Taking either step will ensure that if a transfer fails, the transaction will also fail.

Long term, use an automated tool such as [Slither](https://github.com/crytic/slither) to detect other instances of `transfer()` calls with unchecked return values.

26. Differences between public repository, deployed contracts, and private repository

Severity: Informational
Difficulty: Low
Type: Configuration
Finding ID: TOB-FRAX-026
Target: `Throughout`

Description
The Frax Finance team provided Trail of Bits with a list of deployed contract addresses to review in addition to commits `3f0993` and `6e0352` of the `FraxFinance/frax-solidity` repository. The Frax Finance team also mentioned a second private repository. During our review, we identified discrepancies between certain deployed contracts and the versions of those contracts in the repository. This resulted in cross-checking overhead.

Exploit Scenario
Reviewer Alice performs an audit of the file Frax.sol at commit 3f0993. It differs from the version of the contract deployed on Etherscan. As a result, Alice's findings may be outdated, or she may overlook issues present in the newer version.

Recommendations
Short term, clearly communicate the differences between the public repository and the deployed contracts to auditors. If possible, instruct auditors to work either solely from the contracts deployed on Etherscan or solely from the repository.

Long term, maintain consistency between deployed contracts and their files in the Git repository. Each time a contract is deployed to the mainnet, "freeze" the file and its dependencies in GitHub. Instead of modifying the file of the deployed version, create a copy with a suffix (e.g., V2, V3, etc.) and work on that version until it is deployed to the mainnet. Then, repeat the process. This will simplify future reviews and increase their precision.

27. Owners and governance can set fees and other parameters to any value

Severity: Undetermined
Type: Data Validation
Target: `Frax.sol`

Difficulty: Low
Finding ID: TOB-FRAX-027

Description
Owners and governance can set the `minting_fee`, the `redemption_fee`, and other system parameters to any value, even very large values that could have unexpected effects on the system.

Exploit Scenario
Governance sets the `minting_fee` to a very large value. As part of the minting process, the high minting fee is subtracted from the amount to be paid out. This causes an underflow, and the use of `SafeMath` causes a revert. As a result, minting is no longer possible.

Recommendations
Short term, establish reasonable upper bounds for fees and implement checks to ensure that the fees do not exceed them.

Long term, establish reasonable lower and upper bounds for all system parameters and implement a method of validating them. This will prevent system participants from accidentally or maliciously forcing the system into an unexpected or dysfunctional state.

A. Vulnerability Classifications

Vulnerability Classes	
Class	**Description**
Access Controls	Related to authorization of users and assessment of rights
Auditing and Logging	Related to auditing of actions or logging of problems
Authentication	Related to the identification of users
Configuration	Related to security configurations of servers, devices, or software
Cryptography	Related to protecting the privacy or integrity of data
Data Exposure	Related to unintended exposure of sensitive information
Data Validation	Related to improper reliance on the structure or values of data
Denial of Service	Related to causing a system failure
Error Reporting	Related to the reporting of error conditions in a secure fashion
Patching	Related to keeping software up to date
Session Management	Related to the identification of authenticated users
Testing	Related to test methodology or test coverage
Timing	Related to race conditions, locking, or the order of operations
Undefined Behavior	Related to undefined behavior triggered by the program

Severity Categories	
Severity	**Description**
Informational	The issue does not pose an immediate risk but is relevant to security best practices or Defense in Depth.
Undetermined	The extent of the risk was not determined during this engagement.
Low	The risk is relatively small or is not a risk the customer has indicated is important.

Medium	Individual users' information is at risk; exploitation could pose reputational, legal, or moderate financial risks to the client.
High	The issue could affect numerous users and have serious reputational, legal, or financial implications for the client.

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

B. Code Maturity Classifications

Code Maturity Classes	
Category Name	**Description**
Access Controls	Related to the authentication and authorization of components
Arithmetic	Related to the proper use of mathematical operations and semantics
Assembly Use	Related to the use of inline assembly
Centralization	Related to the existence of a single point of failure
Upgradeability	Related to contract upgradeability
Function Composition	Related to separation of the logic into functions with clear purposes
Front-Running	Related to resilience against front-running
Key Management	Related to the existence of proper procedures for key generation, distribution, and access
Monitoring	Related to the use of events and monitoring procedures
Specification	Related to the expected codebase documentation
Testing & Verification	Related to the use of testing techniques (unit tests, fuzzing, symbolic execution, etc.)

Rating Criteria	
Rating	**Description**
Strong	The component was reviewed, and no concerns were found.
Satisfactory	The component had only minor issues.
Moderate	The component had some issues.
Weak	The component led to multiple issues; more issues might be present.
Missing	The component was missing.

Not Applicable	The component is not applicable.
Not Considered	The component was not reviewed.
Further Investigation Required	The component requires further investigation.

C. Code Quality

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

Frax.sol

- **The local variable `eth_usd_price` in line `116` shadows the function `eth_usd_price()`.** Consider renaming it to increase the code's clarity.
- **The local variable `price_vs_eth` in line `136` is not initialized.** Consider initializing it to `0` to increase the code's clarity.

FXS.sol

- **The function `getChainId` is not used.** Removing it would make the code easier to maintain and review.

FraxPool.sol

- **The local variables `FXSAmount` and `CollateralAmount` in the function `collectRedemption` are not initialized.** Consider initializing them to `0` to clarify the code.
- **`if(a == true)` is redundant.** Consider using `if(a)` instead to simplify the code and increase readability.

FXS1559_AMO.sol

- **The use of "`2105300114 // A long time from now`" in two lines may cause confusion.** Consider adding a comment that describes the exact amount of time represented by `2105300114` to increase the code's clarity.

UniswapV2Library.sol

- **The local variable `i` in line `74` is not initialized.** Initializing it to `0` would clarify the code.

Curve/IMetaImplementationUSD.sol

- **The function signature `transfer(address _to, uint _value) external returns (uint256)` is not correct for an ERC20 token.** Consider changing the return type to `bool` to conform to the ERC20 standard.

Curve/IStableSwap3Pool.sol

- **The function signature `transfer(address _to, uint _value) external returns (uint256)` is not correct for an ERC20 token.** Consider changing the return type to `bool` to conform to the ERC20 standard.

Curve/CurveAMO_V3.sol.sol

- **Line `142` uses the address constant `0xB4AdA607B9d6b2c9Ee07A275e9616B84AC560139`, which is already assigned to the variable `crvFRAX_vault_address`.** Consider reusing the variable `crvFRAX_vault_address` to increase readability.
- **`burnFRAX` and `metapoolDeposit` could in theory be reentered from `FRAX`.** This is not currently an issue, as the `FRAX` implementation does not have that capability. Consider changing the order of operations such that it follows the checks-effects-interactions pattern.
- **`giveCollatBACK` could in theory be reentered from `collateral_token`.** Consider changing the order of operations such that it follows the checks-effects-interactions pattern.

ERC20Custom.sol

- **The function `_beforeTokenTransfer` is never used.** Consider removing dead code to make the codebase easier to maintain and review.

ERC20.sol

- **The parameters `name` and `symbol` of the `constructor` shadow the functions that share their names.** Consider renaming them to increase the code's clarity.

Babylonian.sol

- **The return value of the `sqrt` function is implicitly `0` if none of the branches are executed.** Consider setting `z = 0` in an explicit `else` branch to increase the clarity of the code.

InvestorAMO_V2.sol

- **Follow the checks-effects-interaction pattern to ensure that events are not emitted out of order.**
  - `InvestorAMO_V2.emergencyRecoverERC20()`

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. An up-to-date version of the checklist can be found in 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
```

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 Security 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.

ERC Conformity

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.
- ❏ **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.

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]().

Finally, there are certain characteristics that are difficult to identify automatically. Conduct a manual review of the following conditions:

- ❏ **`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.

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 if 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.

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.

E. Risks Associated with Allowing CurveAMO_V3 to Be Publicly Callable

Frax Finance aims to allow `CurveAMO_V3`'s functions to be publicly callable. As such, the development team must consider additional security measures to ensure that these functions cannot be abused. We recommend that Frax Finance analyze the following areas to ensure that all functions have sufficient validation of incoming values. This will prevent attackers from arbitrarily minting and burning pool tokens when making deposits. Frax Finance should take the following steps:

- **Add clear documentation on functions that are expected to be public.** All functions that are intended to be publicly callable in the future should be clearly identified, and the risks of that configuration should be analyzed throughout the development process.
- **Carefully validate all inputs provided by callers, especially when funds are minted or burned from a pool.** This will ensure that third parties cannot maliciously affect the minting and burning of FRAX or FXS, which could have implications for the price stability of FRAX. The following variables require special attention:
    - `mintRedeemPart1 - frax_amount`
    - `burnFXS - amount`
    - `burnFRAX - frax_amount`
    - `giveCollatBack - amount`
    - `metapoolDeposit - _frax_amount, _collateral_amount`
    - `metapoolWithdrawAtCurRatio - _metapool_lp_in`
    - `metapoolWithdrawFrax - _metapool_lp_in`
    - `three_pool_to_collateral  _3pool_in`
- **Implement an upper bound on slippage values.** Third parties should not be able to force trades with very high slippage.
- **Identify and document system parameters.** Every parameter should be documented for end users.

F. Fix Log

On the week of June 7, 2021, Trail of Bits reviewed fixes for issues identified in this report. The Frax Finance team fixed nine issues and partially fixed four issues reported in the original assessment. We reviewed each of the fixes to ensure that the proposed remediation would be effective.

ID	Title	Severity	Status
1	Transfers of collateral tokens can silently fail, causing a loss of funds	High	Fixed (60)
2	Lack of two-step process for critical operations	High	Fixed (60)
3	Missing events for critical operations	Informational	Partially Fixed (60)
4	Inconsistent use of the term “governance”	Informational	Fixed (67)
5	Lack of zero check on functions	Informational	Partially Fixed (60)
6	Lack of return value check in veFXS may result in failed ERC20 token recovery	Low	Not Fixed
7	Lack of return value check in FXS may result in unexpected behavior	Informational	Fixed (60)
8	Solidity compiler optimizations can be problematic	Informational	Not Fixed
9	Initialization functions can be front-run	High	Fixed
10	Lack of return value check in CurveAMO_V3 may result in failed collateral retrieval	High	Fixed (60)
11	Lack of contract existence check on delegatecall will result in unexpected behavior	High	Not Fixed
12	Aragon’s voting contract does not follow voting best practices	High	Partially Fixed
13	Two-block delay may not deter whale activity	Informational	Not Fixed
14	Ether can be deposited into CurveAMO_V3 but not retrieved from it	Low	Fixed (60)

15	Contracts used as dependencies do not track upstream changes	Low	Not Fixed
16	External calls in loops may result in denial of service	Informational	Not Fixed
17	Lack of contract and user documentation	Informational	Not Fixed
18	Use of Solidity arithmetic may result in integer overflows	Informational	Fixed (60)
19	Curve AMO assumes the collateral ratio to be constant	Undetermined	Not Fixed
20	isContract() may behave unexpectedly	Informational	Not Fixed
21	Risks related to CurveDAO architecture	High	Not Fixed
22	Pool deployment will fail if collateral token has more than 18 decimals	Low	Not Fixed
23	One-to-one minting and redeeming operations have different collateral ratio requirements	Undetermined	Not Fixed
24	Use of non-production-ready ABIEncoderV2	Undetermined	Partially Fixed (60)
25	Lack of return value check in Investor AMO contract	Low	Fixed (60, 68)
26	Differences between public repository, deployed contracts, and private repository	Informational	Not Fixed
27	Owners and governance can set fees and other parameters to any value	Undetermined	Not Fixed

Detailed Fix Log

Finding 1. Transfers of collateral tokens can silently fail, causing a loss of funds

Fixed. The functions that silently failed were replaced with `TransferHelper.safeTransfer` and `TransferHelper.safeTransferFrom`. As a result, transactions will revert upon failed transfers.

Finding 2. Lack of two-step process for critical operations

Fixed. The `FraxPool`, `FXS`, and `FRAX` contracts were refactored to use Synthetix's [`Owned`](), and the `CurveAMO_V3` contract was refactored to use Synthetix's [`Owned_Proxy`](). As stated in [TOB-FRAX-015](), we highly recommend developing additional documentation specifying the commit of each dependency and/or providing source code links.

Finding 3. Missing events for critical operations

Partially fixed. Events were added for the majority of the operations that lacked them, with the exception of those in the `CurveAMO_V3` contract and the FXS' `toggleVote` function.

Frax Finance stated the following:

> *Added, except for AMOs*

Finding 5. Lack of zero check on functions

Partially fixed. Zero checks were added to many of the functions but are still missing from `FraxPool.setCollatETHOracle`, `FraxPool.setTimeLock`, and `Frax.setPriceBand`.

Finding 6. Lack of return value check in veFXS may result in failed ERC20 token recovery

Not fixed. Frax Finance stated the following:

> *Skipped, as it is a corner case / governance only*

Finding 7. Lack of return value check in FXS may result in unexpected behavior

Fixed. The functions that lacked return value checks were replaced by `TransferHelper.safeTransfer` and `TransferHelper.safeTransferFrom`. As a result, transactions will revert upon failed transfers.

Finding 8. Solidity compiler optimizations can be problematic

Not fixed. The issue was *"skipped,"* according to Frax Finance.

Finding 9. Initialization functions can be front-run

Fixed. Frax Finance stated that it is using `hardhat-upgrades` to set up the proxy and to connect the implementations:

We are using hardhat-upgrades actually

However, Trail of Bits did not review these deployment scripts.

Finding 10. Lack of return value check in CurveAMO_V3 may result in failed collateral retrieval

Fixed. The functions that lacked return value checks were replaced by `TransferHelper.safeTransfer` and `TransferHelper.safeTransferFrom`. As a result, transactions will revert upon failing.

Finding 11. Lack of contract existence check on delegatecall will result in unexpected behavior

Not fixed. Frax Finance uses `hardhat-upgrades` to update the implementation contract. However, the lack of a contract existence check in the code can still cause undefined behavior if the target implementation contract is self-destructed after it is updated.

Reference: https://blog.trailofbits.com/2020/12/16/breaking-aave-upgradeability/

Finding 12. Aragon's voting contract does not follow voting best practices

Partially fixed. Frax Finance stated the following:

veFXS itself has locked FXS. We are also using Snapshot for the moment

However, the respective contract still contains Aragon-specific methods for backward compatibility. Trail of Bits recommends either removing the Aragon-specific code or adding documentation outlining the future plans for the voting implementation.

Finding 13. Two-block delay may not deter whale activity

Not fixed. Frax Finance stated the following:

Cannot prevent whales

Finding 15. Contracts used as dependencies do not track upstream changes

Not fixed. Frax Finance stated the following:

Becomes difficult because of hardhat and compiling with mixed versions (pragma solidity X.Y.Z)

Finding 16. External calls in loops may result in denial of service

Not fixed. Frax Finance stated the following:

I don't think the loops are avoidable, but we are aware that loops can be gassy and/or problematic and will try to limit them

Finding 17. Lack of contract and user documentation

Not fixed. Frax Finance stated the following:

> *We will update comments as we gradually move to a more decentralized model and some of these 'setter' functions become publically callable 'refresh' functions that have algorithms that determine state values.*

Finding 19. Curve AMO assumes the collateral ratio to be constant

Not fixed. Frax Finance stated the following:

> *Will document when we move to be more decentralized / have an algorithm here*

Finding 20. isContract() may behave unexpectedly

Not fixed. Frax Finance stated the following:

> *Noted, but will leave for now*

Finding 21. Risks related to CurveDAO architecture

Not fixed. Frax Finance stated the following:

> *We cannot control this, and it is a known risk*

Finding 22. Pool deployment will fail if collateral token has more than 18 decimals

Not fixed. Frax Finance stated the following:

> *We don't ever anticipate any tokens above 18 decimals, and never actually encountered one.*

Finding 23. One-to-one minting and redeeming operations have different collateral ratio requirements

Not fixed. Frax Finance stated the following:

> *This is known and by design*

Finding 24. Use of non-production-ready ABIEncoderV2

Partially fixed. Frax Finance stated the following:

> *Removed in a few places, but a lot of the staking contracts still need it.*

Finding 25. Lack of return value check in Investor AMO contract

Fixed. The functions that lacked return value checks were replaced by `TransferHelper.safeTransfer` and `TransferHelper.safeTransferFrom`. As a result, transactions will revert upon failing.

Finding 26. Differences between public repository, deployed contracts, and private repository

Not fixed. Frax Finance *"noted"* the issue.

Finding 27. Owners and governance can set fees and other parameters to any value
Not fixed. Frax Finance stated the following:

> *Will change when we become more decentralized*