Frax Finance

Security Assessment

August 3, 2022

Prepared for:
Sam Kazemian
Frax Finance

Prepared by: **Gustavo Grieco and Michael Colburn**

About Trail of Bits

Founded in 2012 and headquartered in New York, Trail of Bits provides technical security assessment and advisory services to some of the world's most targeted organizations. We combine high-end security research with a real-world attacker mentality to reduce risk and fortify code. With 80+ employees around the globe, we've helped secure critical software elements that support billions of end users, including Kubernetes and the Linux kernel.

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

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

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

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

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

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

Notices and Remarks

Copyright and Distribution

Test Coverage Disclaimer

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

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

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

Table of Contents

Executive Summary

Engagement Overview

Frax Finance engaged Trail of Bits to review the security of the FraxSwap, `FPIController`, and FraxLend contracts. From April 25 to May 5, 2022, a team of two consultants conducted a security review of the client-provided source code, with four person-weeks of effort. Details of the project's timeline, test targets, and coverage are provided in subsequent sections of this report.

Project Scope

Our testing efforts were focused on the identification of flaws that could result in a compromise of confidentiality, integrity, or availability of the target system. We conducted this audit with full knowledge of the target system, including access to the source code and documentation. We performed static and dynamic testing of the target system and its codebase, using both automated and manual processes.

Summary of Findings

The audit uncovered significant flaws that could impact system integrity or availability. A summary of the findings and details on notable findings are provided below.

EXPOSURE ANALYSIS

Severity	*Count*
High	4
Medium	1
Low	1

CATEGORY BREAKDOWN

Category	*Count*
Access Controls	1
Auditing and Logging	1
Data Validation	4

Notable Findings

Significant flaws that impact system confidentiality, integrity, or availability are listed below.

- **TOB-FRAXSWAP-2**
  The code does not check the available liquidity before executing long-term swaps. Long-term swaps executed in the context of insufficient liquidity could cause all operations, including those for adding liquidity, to be blocked.

- **TOB-FRAXFPI-4**
  Explicit integer conversions can be used to bypass certain restrictions in the `FPIControllerPool` contract.

- **TOB-FRAXSWAP-5**
  Some FraxLend functions do not update the exchange rate, allowing insolvent users to call them.

Project Summary

Contact Information

The following managers were associated with this project:

Dan Guido, Account Manager
dan@trailofbits.com

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

The following engineers were associated with this project:

Gustavo Grieco, Consultant
gustavo.grieco@trailofbits.com

Michael Colburn, Consultant
michael.colburn@trailofbits.com

Project Timeline

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

Date	Event
April 21, 2022	Pre-project kickoff call
May 2, 2022	Status update meeting #1
May 9, 2022	Delivery of report draft
May 9, 2022	Report readout meeting
June 23, 2022	Delivery of final report
August 3, 2022	Delivery of publication-ready report

Project Goals

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

- Are proper access controls used in the contracts that users interact with?
- Could an unauthorized user mint or borrow shares?
- Could one user withdraw another user's tokens?
- Are any of the components vulnerable to price manipulation?
- Could an attacker realize a profit through illicit arbitrage?
- Could a user's funds be frozen or stuck?
- Are events used appropriately?

Project Targets

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

`frax-solidity`

Repository	https://github.com/FraxFinance/frax-solidity/
Version	`19dd2637ba17b8224173fd05ff356a58706edb82`
Type	Solidity
Platform	EVM

`frax-lend`

Repository	https://github.com/FraxFinance/frax-lend/
Version	`ca1a39f5ad35a5a470a3eabf7df35a883d92240c`
Type	Solidity
Platform	EVM

Project Coverage

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

- **FraxSwap**: This is an automated market maker (AMM) based on Uniswap V2. It allows users to make not only instantaneous swaps between tokens but also long-term swaps that will be executed for every operation that they perform (e.g., adding or removing liquidity). We manually reviewed the FraxSwap contracts to ensure that all of the bookkeeping is consistent and that the calculations are sound. We also used smart contract fuzzing to ensure that important invariants hold.

- **`FPIController` and `CPITrackerOracle`**: These components are designed to ensure that the Frax Price Index (FPI) targets the Consumer Price Index (CPI) peg. We manually checked them for logic errors, missing access controls, and incorrect arithmetic computations of their internal variables.

- **FraxLend**: This is a lending platform that allows users who remain solvent to deposit collateral and borrow funds. Any user can liquidate the collateral of users who become insolvent. We manually reviewed the FraxLend contracts to ensure that users are not allowed to borrow more than expected or interfere with the internal bookkeeping of the platform.

Coverage Limitations

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

- We did not review the FraxSwap, `FPIController`, `CPITrackerOracle`, and FraxLend contracts' external interactions with other Frax Finance components or arbitrary ERC20 tokens.

- We did not extensively search for front-running vulnerabilities.

- We did not review the high-level economic incentives and disincentives imposed by the system.

Automated Testing

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

Test Harness Configuration

We used Echidna, our smart contract fuzzer that can rapidly test security properties via malicious, coverage-guided test case generation, in the automated testing phase of this project. Our automated testing work focused on identifying issues related to FraxSwap invariants, liquidity/balance checks, and unexpected reverts.

Test Results

The results of this focused testing are detailed below.

Property	Result
If enough liquidity is provided, the amounts specified in transactions cannot be lower than the minimum value.	Passed
If the preconditions are met, transactions for adding liquidity never revert.	TOB-FRAXSWAP-2
If transactions for adding liquidity succeed, token balances after such transactions are in sync with the liquidity state variables.	Passed
Transactions for adding liquidity never result in a decrease of `reserve0 * reserve1` (K).	Passed
If the preconditions are met, transactions for adding liquidity never revert.	TOB-FRAXSWAP-2
Transactions for removing liquidity never result in an increase of `reserve0 * reserve1` (K).	Passed
If a transaction for adding liquidity succeeds, token balances after such	Passed

transactions are in sync with the liquidity state variables.	
If the preconditions are met, calls to `skim` never revert.	TOB-FRAXSWAP-2
If the input passed to `skim` is not the address of the contract itself, token balances are in sync with the liquidity state variables.	Passed
If the preconditions are met, swapping operations never revert.	TOB-FRAXSWAP-2
Swapping operations never decrease `reserve0 * reserve1` (K).	Passed
Transactions for creating long-term swaps with an initial amount of zero always revert.	Passed
If the preconditions are met, operations for canceling orders never revert and always set the orders as complete.	TOB-FRAXSWAP-2
When an order is canceled, the unsold amount cannot be more than the initial amount.	Passed
If the preconditions are met, transactions for withdrawing from long-term swaps never revert.	TOB-FRAXSWAP-2

Note that the issue described in TOB-FRAXSWAP-2 caused the FraxSwap contract to enter an invalid state, which prevented the fuzzing campaign from continuing. Some of the properties marked with “Passed” above should be explored further after TOB-FRAXSWAP-2 is fixed to ensure that the code works as expected.

Codebase Maturity Evaluation

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

Category	Summary	Result
Arithmetic	The codebase contains numerous numerical computations, most of which are performed with safe arithmetic. However, some parts of the code contain unchecked arithmetic operations that are not properly documented. These parts of the code will need extensive testing to ensure that they do not overflow in a way that affects the system's invariants. Additionally, the system's use of reverts to handle overflows may block important operations and should be reconsidered (TOB-FRAXSWAP-2).	**Weak**
Auditing	Many critical administrative functions do not emit events on important state changes (TOB-FRAXSWAP-3). This makes off-chain monitoring difficult to conduct.	**Weak**
Authentication / Access Controls	While the in-scope components' access controls are sufficient, the roles and responsibilities of all parties should be clarified, as explained in TOB-FRAXLEND-6.	**Moderate**
Complexity Management	The functions and contracts are organized and scoped appropriately and contain inline documentation that explains their workings. Though there is a clear separation of duties between each of the core contracts in the system, this can cause some interactions to be difficult to follow as they pass between different contracts.	**Moderate**
Decentralization	While the in-scope components were designed to be decentralized and have privileged operations that are	**Further Investigation**

	accessible only through the governance process, the interactions with other contracts should be reviewed in depth to ensure that they cannot introduce centralization issues.	**Required**
Documentation	The project has good high-level documentation, a specification indicating the derivations for the formulas used by the system, and good use of NatSpec and inline comments. However, the system's invariants are not specified.	**Satisfactory**
Front-Running Resistance	AMMs and lending platforms are vulnerable to front-running issues; the codebase should undergo a more in-depth review to find such vulnerabilities.	**Further Investigation Required**
Testing and Verification	The codebase contains a number of unit and integration tests. However, the tests are insufficient to catch high-severity issues, which indicates that the test suite should be improved. Moreover, the codebase would benefit from advanced testing techniques such as fuzzing.	**Weak**

Summary of Findings

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

ID	Title	Type	Severity
1	Risk of unexpected results when long-term swaps involving rebasing tokens are canceled	Data Validation	High
2	Missing liquidity checks when initiating long-term swaps	Data Validation	High
3	Missing events in several contracts	Auditing and Logging	Low
4	Unsafe integer conversions in FPIControllerPool	Data Validation	High
5	leveragedPosition and repayAssetWithCollateral do not update the exchangeRate	Data Validation	High
6	Risk of hash collisions in FraxLendPairDeployer that could block certain deployments	Access Controls	Medium

Detailed Findings

1. Risk of unexpected results when long-term swaps involving rebasing tokens are canceled	
Severity: **High**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-FRAXSWAP-1
Target: `UniV2TWAMMPair.sol`	

Description

FraxSwap's use of rebasing tokens—tokens whose supply can be adjusted to control their prices—could cause transactions to revert after users cancel or withdraw from long-term swaps.

FraxSwap offers a new type of swap called a "long-term swap," which executes certain swaps over an extended period of time. Users can cancel or withdraw from long-term swaps and recover all of their purchased and unsold tokens.

```
    ///@notice stop the execution of a long term order
     function cancelLongTermSwap(uint256 orderId) external lock execVirtualOrders {
         (address sellToken, uint256 unsoldAmount, address buyToken, uint256
purchasedAmount) = longTermOrders.cancelLongTermSwap(orderId);

         bool buyToken0 = buyToken == token0;
         twammReserve0 -= uint112(buyToken0 ? purchasedAmount : unsoldAmount);
         twammReserve1 -= uint112(buyToken0 ? unsoldAmount : purchasedAmount);

         // transfer to owner of order
         _safeTransfer(buyToken, msg.sender, purchasedAmount);
         _safeTransfer(sellToken, msg.sender, unsoldAmount);

         // update order. Used for tracking / informational
         longTermOrders.orderMap[orderId].isComplete = true;

         emit CancelLongTermOrder(msg.sender, orderId, sellToken, unsoldAmount,
buyToken, purchasedAmount);
     }
```

Figure 1.1: The `cancelLongTermSwap` function in the `UniV2TWAMMPair` contract

However, if a rebasing token is used in a long-term swap, the balance of the `UniV2TWAMMPair` contract could increase or decrease over time. Such changes in the contract's balance could result in unintended effects when users try to cancel or withdraw from long-term swaps. For example, because all long-term swaps for a pair are processed as part of any function with the `execVirtualOrders` modifier, if the actual balance of the `UniV2TWAMMPair` is reduced as part of one or more rebases in the underlying token, this balance will not be reflected correctly in the contract's internal accounting, and cancel and withdraw operations will transfer too many tokens to users. Eventually, this will exhaust the contract's balance of the token before all users are able to withdraw, causing these transactions to revert.

Exploit Scenario

Alice creates a long-term swap; one of the tokens to be swapped is a rebasing token. After some time, the token's supply is adjusted, causing the balance of `UniV2TWAMMPair` to decrease. Alice tries to cancel the long-term swap, but the internal bookkeeping for her swap was not updated to reflect the rebase, causing the token transfer from the contract to Alice to revert and blocking her other token transfers from completing. To allow Alice to access funds and to allow subsequent transactions to succeed, some tokens need to be explicitly sent to the `UniV2TWAMMPair` contract to increase its balance.

Recommendations

Short term, explicitly document issues involving rebasing tokens and long-term swaps to ensure that users are aware of them.

Long term, evaluate the security risks surrounding ERC20 tokens and how they could affect every system component.

References

- Common errors with rebasing tokens on Uniswap V2

2. Missing liquidity checks when initiating long-term swaps	
Severity: **High**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-FRAXSWAP-2
Target: `UniV2TWAMMPair.sol`, `LongTermOrders.sol`, `ExecVirtualOrders.sol`	

Description

When a long-term swap is submitted to a `UniV2TWAMMPair` instance, the code performs checks, such as those ensuring that the selling rate of a given token is nonzero, before the order is recorded. However, the code does not validate the existing reserves for the tokens being bought in long-term swaps.

```
    ///@notice adds long term swap to order pool
    function performLongTermSwap(LongTermOrders storage longTermOrders, address
from, address to, uint256 amount, uint256 numberOfTimeIntervals) private returns
(uint256) {
        // make sure to update virtual order state (before calling this function)

        //determine the selling rate based on number of blocks to expiry and total
amount
        uint256 currentTime = block.timestamp;
        uint256 lastExpiryTimestamp = currentTime - (currentTime %
longTermOrders.orderTimeInterval);
        uint256 orderExpiry = longTermOrders.orderTimeInterval *
(numberOfTimeIntervals + 1) + lastExpiryTimestamp;
        uint256 sellingRate = SELL_RATE_ADDITIONAL_PRECISION * amount / (orderExpiry
- currentTime);

        require(sellingRate > 0); // tokenRate cannot be zero
```

Figure 2.1: `Uniswap_V2_TWAMM/twamm/LongTermOrders.sol#L118-L128`

If a long-term swap is submitted before adequate liquidity has been added to the pool, the next pool operation will attempt to trade against inadequate liquidity, resulting in a division-by-zero error in the line highlighted in blue in figure 2.2. As a result, all pool operations will revert until the number of tokens needed to begin executing the virtual swap are added.

```
    ///@notice computes the result of virtual trades by the token pools
    function computeVirtualBalances(
        uint256 token0Start,
        uint256 token1Start,
        uint256 token0In,
        uint256 token1In)
    internal pure returns (uint256 token0Out, uint256 token1Out, uint256
ammEndToken0, uint256 ammEndToken1)
    {
        token0Out = 0;
        token1Out = 0;
        //when both pools sell, we use the TWAMM formula
        else {
            uint256 aIn = token0In * 997 / 1000;
            uint256 bIn = token1In * 997 / 1000;
            uint256 k = token0Start * token1Start;
            ammEndToken1 = token0Start * (token1Start + bIn) / (token0Start + aIn);
            ammEndToken0 = k / ammEndToken1;
            token0Out = token0Start + aIn - ammEndToken0;
            token1Out = token1Start + bIn - ammEndToken1;
        }
```

Figure 2.2: `Uniswap_V2_TWAMM/twamm/ExecVirtualOrders.sol#L39-L78`

The long-term swap functionality can be paused by the contract owner (e.g., to prevent long-term swaps when a pool has inadequate liquidity); however, by default, the functionality is enabled when a new pool is deployed. An attacker could exploit this fact to grief a newly deployed pool by submitting long-term swaps early in its lifecycle when it has minimal liquidity.

Additionally, even if a newly deployed pool is already loaded with adequate liquidity, a user could submit long-term swaps with zero intervals to trigger an integer underflow in the line highlighted in red in figure 2.2. However, note that the user would have to submit at least one long-term swap that requires more than the total liquidity in the reserve:

```
testSync(): failed!💥
Call sequence:
    ●   initialize(6809753114178753104760,5497681857357274469621,8379829307706602317717,10991961728915299510446)
    ●   longTermSwapFrom1To0(2,0)
    ●   testLongTermSwapFrom0To1(234162462256667058826000049678018899445043512014876676541160061227714669,0)
    ●   testSync() Time delay: 37073 seconds Block delay: 48
```

Figure 2.3: The Echidna output that triggers a revert in a call to `sync()`

Exploit Scenario

A new FraxSwap pool is deployed, causing the long-term swap functionality to be unpaused. Before users have a chance to add sufficient liquidity to the pool, Eve initiates long-term swaps in both directions. Since there are no tokens available to purchase, all pool operations revert, and the provided tokens are trapped. At this point, adding more liquidity is not possible since doing so also triggers the long-term swap computation, forcing a revert.

Recommendations

Short term, disable new swaps and remove all the liquidity in the deployed contracts. Modify the code so that, moving forward, liquidity can be added without executing long-term swaps. Document the pool state requirements before long-term swaps can be enabled.

Long term, use extensive smart contract fuzzing to test that important operations cannot be blocked.

3. Missing events in several contracts	
Severity: **Low**	Difficulty: **Low**
Type: Auditing and Logging	Finding ID: TOB-FRAXSWAP-3
Target: `UniV2TWAMMPair.sol`, `FPIControllerPool.sol`, `CPITrackerOracle.sol`, and dependencies	

Description

An insufficient number of events is declared in the Frax Finance contracts. As a result, malfunctioning contracts or malicious attacks may not be noticed.

For instance, long-term swaps are executed in batches by the `executeVirtualOrdersUntilTimestamp` function:

```
    ///@notice executes all virtual orders until blockTimestamp is reached.
    function executeVirtualOrdersUntilTimestamp(LongTermOrders storage
longTermOrders, uint256 blockTimestamp, ExecuteVirtualOrdersResult memory
reserveResult) internal {
        uint256 nextExpiryBlockTimestamp = longTermOrders.lastVirtualOrderTimestamp
- (longTermOrders.lastVirtualOrderTimestamp % longTermOrders.orderTimeInterval) +
longTermOrders.orderTimeInterval;
        //iterate through time intervals eligible for order expiries, moving state
forward

        OrderPool storage orderPool0 = longTermOrders.OrderPool0;
        OrderPool storage orderPool1 = longTermOrders.OrderPool1;

        while (nextExpiryBlockTimestamp < blockTimestamp) {
            // Optimization for skipping blocks with no expiry
            if (orderPool0.salesRateEndingPerTimeInterval[nextExpiryBlockTimestamp]
> 0
                ||
orderPool1.salesRateEndingPerTimeInterval[nextExpiryBlockTimestamp] > 0) {

                //amount sold from virtual trades
                uint256 blockTimestampElapsed = nextExpiryBlockTimestamp -
longTermOrders.lastVirtualOrderTimestamp;
                uint256 token0SellAmount = orderPool0.currentSalesRate *
blockTimestampElapsed / SELL_RATE_ADDITIONAL_PRECISION;
                uint256 token1SellAmount = orderPool1.currentSalesRate *
blockTimestampElapsed / SELL_RATE_ADDITIONAL_PRECISION;

                (uint256 token0Out, uint256 token1Out) =
executeVirtualTradesAndOrderExpiries(reserveResult, token0SellAmount,
token1SellAmount);
```

```
                updateOrderPoolAfterExecution(longTermOrders, orderPool0,
orderPool1, token0Out, token1Out, nextExpiryBlockTimestamp);

            }
            nextExpiryBlockTimestamp += longTermOrders.orderTimeInterval;
        }
        //finally, move state to current blockTimestamp if necessary
        if (longTermOrders.lastVirtualOrderTimestamp != blockTimestamp) {

            //amount sold from virtual trades
            uint256 blockTimestampElapsed = blockTimestamp -
longTermOrders.lastVirtualOrderTimestamp;
            uint256 token0SellAmount = orderPool0.currentSalesRate *
blockTimestampElapsed / SELL_RATE_ADDITIONAL_PRECISION;
            uint256 token1SellAmount = orderPool1.currentSalesRate *
blockTimestampElapsed / SELL_RATE_ADDITIONAL_PRECISION;

            (uint256 token0Out, uint256 token1Out) =
executeVirtualTradesAndOrderExpiries(reserveResult, token0SellAmount,
token1SellAmount);
            updateOrderPoolAfterExecution(longTermOrders, orderPool0, orderPool1,
token0Out, token1Out, blockTimestamp);

        }
    }
```

Figure 3.1: `Uniswap_V2_TWAMM/twamm/LongTermOrders.sol#L216-L252`

However, despite the complexity of this function, it does not emit any events; it will be difficult to monitor issues that may arise whenever the function is executed.

Additionally, important operations in the `FPIControllerPool` and `CPITrackerOracle` contracts do not emit any events:

```
    function toggleMints() external onlyByOwnGov {
        mints_paused = !mints_paused;
    }

    function toggleRedeems() external onlyByOwnGov {
        redeems_paused = !redeems_paused;
    }

    function setFraxBorrowCap(int256 _frax_borrow_cap) external onlyByOwnGov {
        frax_borrow_cap = _frax_borrow_cap;
    }

    function setMintCap(uint256 _fpi_mint_cap) external onlyByOwnGov {
        fpi_mint_cap = _fpi_mint_cap;
    }
```

Figure 3.2: `FPI/FPIControllerPool.sol#L528-L542`

Events generated during contract execution aid in monitoring, baselining of behavior, and detection of suspicious activity. Without events, users and blockchain-monitoring systems cannot easily detect behavior that falls outside the baseline conditions, and it will be difficult to review the correct behavior of the contracts once they have been deployed.

Exploit Scenario

Eve, a malicious user, discovers a vulnerability that allows her to manipulate long-term swaps. Because no events are generated from her actions, the attack goes unnoticed. Eve uses her exploit to drain liquidity or prevent other users from swapping before the Frax Finance team has a chance to respond.

Recommendations

Short term, emit events for all operations that may contribute to a higher level of monitoring and alerting, even internal ones.

Long term, consider using a blockchain-monitoring system to track any suspicious behavior in the contracts. A monitoring mechanism for critical events could quickly detect system compromises.

4. Unsafe integer conversions in FPIControllerPool	
Severity: **High**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-FRAXFPI-4
Target: `FPIControllerPool.sol`	

Description

Explicit integer conversions can be used to bypass certain restrictions (e.g., the borrowing cap) in the `FPIControllerPool` contract.

The `FPIControllerPool` contract allows certain users to either borrow or repay FRAX within certain limits (e.g., the borrowing cap):

```
    // Lend the FRAX collateral to an AMO
    function giveFRAXToAMO(address destination_amo, uint256 frax_amount) external
onlyByOwnGov validAMO(destination_amo) {
        int256 frax_amount_i256 = int256(frax_amount);

        // Update the balances first
        require((frax_borrowed_sum + frax_amount_i256) <= frax_borrow_cap, "Borrow
cap");
        frax_borrowed_balances[destination_amo] += frax_amount_i256;
        frax_borrowed_sum += frax_amount_i256;

        // Give the FRAX to the AMO
        TransferHelper.safeTransfer(address(FRAX), destination_amo, frax_amount);
    }

    // AMO gives back FRAX. Needed for proper accounting
    function receiveFRAXFromAMO(uint256 frax_amount) external validAMO(msg.sender) {
        int256 frax_amt_i256 = int256(frax_amount);

        // Give back first
        TransferHelper.safeTransferFrom(address(FRAX), msg.sender, address(this),
frax_amount);

        // Then update the balances
        frax_borrowed_balances[msg.sender] -= frax_amt_i256;
        frax_borrowed_sum -= frax_amt_i256;
    }
```

Figure 4.1: The `giveFRAXToAMO` function in `FPIControllerPool.sol`

However, these functions explicitly convert these variables from `uint256` to `int256`; these conversions will never revert and can produce unexpected results. For instance, if `frax_amount` is set to a very large unsigned integer, it could be cast to a negative number. Malicious users could exploit this fact by adjusting the variables to integers that will bypass the limits imposed by the code after they are cast.

The same issue affects the implementation of `price_info`:

```
    // Get additional info about the peg status
    function price_info() public view returns (
        int256 collat_imbalance,
        uint256 cpi_peg_price,
        uint256 fpi_price,
        uint256 price_diff_frac_abs
    ) {
        fpi_price = getFPIPriceE18();
        cpi_peg_price = cpiTracker.currPegPrice();
        uint256 fpi_supply = FPI_TKN.totalSupply();

        if (fpi_price > cpi_peg_price){
            collat_imbalance = int256(((fpi_price - cpi_peg_price) * fpi_supply) /
PRICE_PRECISION);
            price_diff_frac_abs = ((fpi_price - cpi_peg_price) * PEG_BAND_PRECISION)
/ fpi_price;
        }
        else {
            collat_imbalance = -1 * int256(((cpi_peg_price - fpi_price) *
fpi_supply) / PRICE_PRECISION);
            price_diff_frac_abs = ((cpi_peg_price - fpi_price) * PEG_BAND_PRECISION)
/ fpi_price;
        }
    }
```

Figure 4.2: The `price_info` function in `FPIControllerPool.sol`

Exploit Scenario

Eve submits a governance proposal that can increase the amount of FRAX that can be borrowed. The voters approve the proposal because they believe that the borrowing cap will stop Eve from changing it to a larger value.

Recommendations

Short term, add checks to the relevant functions to validate the results of explicit integer conversions to ensure that they are within the expected range.

Long term, use extensive smart contract fuzzing to test that system invariants cannot be broken.

5. leveragedPosition and repayAssetWithCollateral do not update the exchangeRate

Severity: **High**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-FRAXLEND-5
Target: `FraxLendCore.sol`	

Description

Some FraxLend functions do not update the exchange rate, allowing insolvent users to call them.

The FraxLend platform offers various operations, such as `leveragedPosition` and `repayAssetWithCollateral`, for users to borrow funds as long as they are solvent. The solvency check is implemented by the `isSolvent` modifier, which runs at the end of such operations:

```
    /// @notice Checks for solvency AFTER executing contract code
    /// @param _borrower The borrower whose solvency we will check
    modifier isSolvent(address _borrower) {
        _;
        require(_isSolvent(_borrower, exchangeRateInfo.exchangeRate), "FraxLendPair:
user is insolvent");
    }
```

Figure 5.1: The `isSolvent` modifier in the `FraxLendCore` contract

However, this modifier is not enough to ensure solvency since the exchange rate changes over time, which can make previously solvent users insolvent. That is why it is important to force an update of the exchange rate during any operation that allows users to borrow funds:

```
    function updateExchangeRate() public returns (uint256 _exchangeRate) {
        ExchangeRateInfo memory _exchangeRateInfo = exchangeRateInfo;
        if (_exchangeRateInfo.lastTimestamp == block.timestamp) return _exchangeRate
= _exchangeRateInfo.exchangeRate;

        …

        // write to storage
        _exchangeRateInfo.exchangeRate = uint224(_exchangeRate);
        _exchangeRateInfo.lastTimestamp = uint32(block.timestamp);
        exchangeRateInfo = _exchangeRateInfo;
        emit UpdateExchangeRate(_exchangeRate);
    }
```
Figure 5.2: Part of the `updateExchangeRate` function in the `FraxLendCore` contract

However, the `leveragedPosition` and `repayAssetWithCollateral` operations increase the debt of the caller but do not call `updateExchangeRate`; therefore, they will perform the solvency check with old information.

Exploit Scenario
Eve, a malicious user, notices a drop in the collateral price. She calls `leveragedPosition` or `repayAssetWithCollateral` to borrow more than the amount of shares/collateral she should be able to.

Recommendations
Short term, add a call to `updateExchangeRate` in every function that increases users' debt.

Long term, document the invariants and preconditions for every function and use extensive smart contract fuzzing to test that system invariants cannot be broken.

6. Risk of hash collisions in FraxLendPairDeployer that could block certain deployments

Severity: **Medium**	Difficulty: **High**
Type: Access Controls	Finding ID: TOB-FRAXLEND-6
Target: `FraxLendPairDeployer.sol`	

Description

A hash collision could occur in the `FraxLendPairDeployer` contract, allowing unauthenticated users to block the deployment of certain contracts from authenticated users.

The `FraxLendPairDeployer` contract allows any user to deploy certain contracts using the `deploy` function, which creates a contract name based on certain parameters:

```
    function deploy(
        address _asset,
        address _collateral,
        address _oracleTop,
        address _oracleDiv,
        uint256 _oracleNormalization,
        address _rateContract,
        bytes calldata _rateInitCallData
    ) external returns (address _pairAddress) {
        string memory _name = string(
            abi.encodePacked(
                "FraxLendV1-",
                IERC20(_collateral).safeName(),
                "/",
                IERC20(_asset).safeName(),
                " - ",
                IRateCalculator(_rateContract).name(),
                " - ",
                deployedPairsArray.length + 1
            )
        );
        …
```

Figure 6.1: The header of the `deploy` function in the `FraxLendPairDeployer` contract

The `_deploySecond` function creates a hash of this contract name and checks it to ensure that it has not already been deployed:

```
    function _deploySecond(
        string memory _name,
        address _pairAddress,
        address _asset,
        address _collateral,
        uint256 _maxLTV,
        uint256 _liquidationFee,
        address _oracleTop,
        address _oracleDiv,
        uint256 _oracleNormalization,
        address _rateContract,
        bytes memory _rateInitCallData,
        address[] memory _approvedBorrowers,
        address[] memory _approvedLenders
    ) private {
        …
        bytes32 _nameHash = keccak256(bytes(_name));
        require(deployedPairsByName[_nameHash] == address(0), "FraxLendPairDeployer:
Pair name must be unique");
        deployedPairsByName[_nameHash] = _pairAddress;
        …
```

Figure 6.2: Part of the `_deploySecond` function in the `FraxLendPairDeployer` contract

Both authenticated and unauthenticated users can use this code to deploy contracts, but only authenticated users can select any name for contracts they want to deploy.

Additionally, the `_deployFirst` function computes a salt based on certain parameters:

```
    function _deployFirst(
        address _asset,
        address _collateral,
        uint256 _maxLTV,
        uint256 _liquidationFee,
        address _oracleTop,
        address _oracleDiv,
        uint256 _oracleNormalization,
        address _rateContract,
        bytes memory _rateInitCallData,
        bool _isBorrowerWhitelistActive,
        bool _isLenderWhitelistActive
    ) private returns (address _pairAddress) {
        {
            //clones are distinguished by their data
            bytes32 salt = keccak256(
                abi.encodePacked(
                    _asset,
                    _collateral,
                    _maxLTV,
                    _liquidationFee,
                    _oracleTop,
                    _oracleDiv,
                    _oracleNormalization,
                    _rateContract,
                    _rateInitCallData,
                    _isBorrowerWhitelistActive,
                    _isLenderWhitelistActive
                )
            );
            require(deployedPairsBySalt[salt] == address(0), "FraxLendPairDeployer:
Pair already deployed");
            …
```

Figure 6.3: The header of the `_deployFirst` function in the `FraxLendPairDeployer` contract

Again, both authenticated and unauthenticated users can use this code, but some parameters are fixed for unauthorized users (e.g., `_maxLTV` will always be `DEFAULT_MAX_LTV` and cannot be changed).

However, in both cases, a hash collision could block contracts from being deployed. For example, if an unauthenticated user sees an authenticated user's pending transaction to deploy a contract, he could deploy his own contract with a name or parameters that result in a hash collision, preventing the authenticated user's contract from being deployed.

Exploit Scenario

Alice, an authenticated user, starts a custom deployment with certain parameters. Eve, a malicious user, sees Alice's unconfirmed transactions and front-runs them with her own call to deploy a contract with similar parameters. Eve's transaction succeeds, causing Alice's deployment to fail and forcing her to change either the contract's name or the parameters of the call to `deploy`.

Recommendations

Short term, prevent collisions between different types of deployments.

Long term, review the permissions and capabilities of authenticated and unauthenticated users in every component.

Summary of Recommendations

The Frax Finance contracts under audit are a work in progress with multiple planned iterations. Trail of Bits recommends that Frax Finance address the findings detailed in this report and take the following additional steps:

- Immediately pause all the deployed contracts affected by TOB-FRAXSWAP-2, notify the affected users, and remove all the liquidity.
- Specify system invariants throughout the codebase and incorporate Echidna into the development process to perform extensive smart contract fuzzing to test such invariants.
- Review all the uses of explicit integer conversions in the codebase, such as the ones described in TOB-FRAXFPI-4.
- Add events for all important administrative and user operations to ensure that the contracts can be easily monitored.

A. Vulnerability Categories

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

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

Severity Levels

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

Difficulty Levels

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

B. Code Maturity Categories

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

Code Maturity Categories

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

Rating Criteria

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

C. Code Quality Recommendations

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

FraxSwap

- **The hardhat compilation environment specifies the use of numerous versions of the Solidity compiler (from version 0.5.x to 0.8.x) and the Vyper compiler.** Consider reducing the use of different compiler versions to avoid unexpected interactions between older and newer versions of the compilers.

- **`_safeTransfer` and `TransferHelper.safeTransfer` implement the same functionality in `UniV2TWAMMPair`.** Consider unifying them into a single function to reduce the complexity of the codebase and the size of the resulting bytecode.

D. Token Integration Checklist

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

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

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

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

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

General Considerations

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

Contract Composition

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

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

Owner Privileges

- ❏ **The token is not upgradeable.** Upgradeable contracts may change their rules over time. Use Slither's `human-summary` printer to determine whether the contract is upgradeable.

- ❏ **The owner has limited minting capabilities.** Malicious or compromised owners can abuse minting capabilities. Use Slither's `human-summary` printer to review minting capabilities, and consider manually reviewing the code.

- ❏ **The token is not pausable.** Malicious or compromised owners can trap contracts relying on pausable tokens. Identify pausable code by hand.

- ❏ **The owner cannot blacklist the contract.** Malicious or compromised owners can trap contracts relying on tokens with a blacklist. Identify blacklisting features by hand.

- ❏ **The team behind the token is known and can be held responsible for abuse.** Contracts with anonymous development teams or teams that reside in legal shelters require a higher standard of review.

ERC20 Tokens

ERC20 Conformity Checks

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

- ❏ **`Transfer` and `transferFrom` return a boolean.** Several tokens do not return a boolean on these functions. As a result, their calls in the contract might fail.

- ❏ **The `name`, `decimals`, and `symbol` functions are present if used.** These functions are optional in the ERC20 standard and may not be present.

- ❏ **`Decimals` returns a `uint8`.** Several tokens incorrectly return a `uint256`. In such cases, ensure that the value returned is below 255.

- ❏ **The token mitigates the known ERC20 race condition.** The ERC20 standard has a known ERC20 race condition that must be mitigated to prevent attackers from stealing tokens.

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

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

Risks of ERC20 Extensions

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

- ❏ **The token is not an ERC777 token and has no external function call in `transfer` or `transferFrom`.** External calls in the transfer functions can lead to reentrancies.

- ❏ **`Transfer` and `transferFrom` should not take a fee.** Deflationary tokens can lead to unexpected behavior.

- ❏ **Potential interest earned from the token is taken into account.** Some tokens distribute interest to token holders. This interest may be trapped in the contract if not taken into account.

Token Scarcity

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

- ❏ **The supply is owned by more than a few users.** If a few users own most of the tokens, they can influence operations based on the tokens' repartition.

- ❏ **The total supply is sufficient.** Tokens with a low total supply can be easily manipulated.

- ❏ **The tokens are located in more than a few exchanges.** If all the tokens are in one exchange, a compromise of the exchange could compromise the contract relying on the token.

- ❏ **Users understand the risks associated with a large amount of funds or flash loans.** Contracts relying on the token balance must account for attackers with a large amount of funds or attacks executed through flash loans.

- ❏ **The token does not allow flash minting.** Flash minting can lead to substantial swings in the balance and the total supply, which necessitate strict and comprehensive overflow checks in the operation of the token.

E. Fuzzing Frax Finance Smart Contracts

During the audit, we created test cases using Echidna to perform fuzz testing on the Frax Finance codebase. Fuzzing is an essential tool for exploring the input space of software to uncover robustness and correctness issues. We focused our fuzzing efforts on the FraxSwap code, but the tests can be extended to include other parts of the codebase as well.

Setup

To run Echidna on the FraxSwap code, we recommend extracting the specific FraxSwap contracts to run the fuzzing campaign. Normally, Echidna can be run on unmodified smart contracts, but in this case, it is faster to extract the FraxSwap contracts from the codebase and to make some minor modifications for testing:

```
$ ln ../../node_modules/@openzeppelin/ . -s
$ ln ../../node_modules/@uniswap . -s
$ ln ../../node_modules/@chainlink . -s
$ slither-flat . --contract UniV2TWAMMPair
$ mv crytic-export/flattening/UniV2TWAMMPair.sol .
$ rm -Rf crytic-export
```

Figure E.1: Extracting the `UniV2TWAMMPair` contract from the codebase

The resulting Solidity file (`UniV2TWAMMPair.sol`) contains the complete source code required to run a FraxSwap pair.

The contract should be initialized with a pair of ERC20 tokens. To do so, we suggest using OpenZeppelin contracts with checks for a minimum liquidity value and a maximum total supply value (2**112):

```
   function initialize(uint112 _totalSupply0, uint112 _totalSupply1, uint112
liquidity0, uint112 liquidity1) public {
      require(address(t0) == address(0x0));
      totalSupply0 = _totalSupply0;
      totalSupply1 = _totalSupply1;
      require(totalSupply0 >= 4 && totalSupply1 >= 4);

      t0 = new ERC20("token0", "t0", totalSupply0);
      t1 = new ERC20("token1", "t1", totalSupply1);

      uint112 amount0 = totalSupply0 / 4;
      uint112 amount1 = totalSupply1 / 4;

      require(amount0 > 10 * MINIMUM_LIQUIDITY && amount1 > 10 * MINIMUM_LIQUIDITY);

      addedMinLiquidity = true;
      t0.transfer(address(0x10000), amount0);
      t0.transfer(address(0x20000), amount0);
      t0.transfer(address(0x30000), amount0);
      t0.transfer(liquidityProvider, amount0);

      t1.transfer(address(0x10000), amount1);
      t1.transfer(address(0x20000), amount1);
      t1.transfer(address(0x30000), amount1);
      t1.transfer(liquidityProvider, amount1);

      try this.initialize(address(t0), address(t1)) {} catch { assert(false); }

      liquidity0 = uint112(10 * MINIMUM_LIQUIDITY + liquidity0 % (amount0 - 10 *
MINIMUM_LIQUIDITY));
      liquidity1 = uint112(10 * MINIMUM_LIQUIDITY + liquidity1 % (amount1 - 10 *
MINIMUM_LIQUIDITY));

      t0.transferFrom(liquidityProvider, address(this), liquidity0);
      t1.transferFrom(liquidityProvider, address(this), liquidity1);

      this.mint(liquidityProvider);
    }
```

Figure E.2: Initialization of the `UniV2TWAMMPair` contract using some liquidity and supply checks

We disabled the allowance checks in the OpenZeppelin ERC20 code that we used for testing, since they do not add useful constraints to the testing (on the contrary, they slow down the fuzz test).

```
    function transferFrom(
        address from,
        address to,
        uint256 amount
    ) public virtual returns (bool) {
        address spender = msg.sender;
        //_spendAllowance(from, spender, amount);
        _transfer(from, to, amount);
        return true;
    }
```

Figure E.3: Modification of the `transferFrom` function to remove allowance checks

Additionally, we recommend making the following minor modifications to the contracts to improve the effectiveness of the fuzzing campaign:

- Convert external functions to public ones, as shown in figure E.4, to more easily verify preconditions and postconditions.

```
    function testLongTermSwapFrom1To0(uint256 amount1In, uint128
numberOfTimeIntervals) public {
      …
      orderId = longTermSwapFrom1To0(amount1In, numberOfTimeIntervals);
      assert(amount1In > 0);
      …
    }
```

Figure E.4: An example of an external function converted to a public one for testing

- Add additional events for internal code, as described in TOB-FRAXSWAP-3. This will help developers debug the code when an assertion fails.

- Add specific reasons for the code to revert so that the fuzzer can catch and verify specific error conditions:

```
  function testCancelLongTermSwap(uint8 oid) public {
    …
    try this.cancelLongTermSwap(orderId) {}
    catch Error (string memory err) {
       longTermOrders.orderMap[orderId].owner = originalOwner;
       if (keccak256(bytes(err)) == keccak256("owner and amounts are invalid"))
         return;
       if (keccak256(bytes(err)) == keccak256("order cannot be expired"))
         return;


       …
    } catch { assert(false); }
    …
  }
```

Figure E.5: An example of a modification that specifies reasons why the code would revert using `try/catch`

- Certain functions in the FraxSwap code, such as `executeVirtualOrdersUntilTimestampView`, are "view only" and can loop for a large number of iterations, which will slow down the fuzzer. We recommend removing or blacklisting them.

Running a Fuzzing Campaign

Run the following command to start a fuzzing campaign using Echidna:

```
$ echidna-test UniV2TWAMMPair.sol --contract EchidnaTest --test-mode
assertion --corpus-dir corpus --seq-len 50
```

Specify the number of transactions to run during the campaign by adding the `--test-limit` X parameter. This number is 50,000 by default, which is too low for the complexity of FraxSwap.

When the campaign ends (or is terminated using the escape key), Echidna will generate a coverage report in the corpus directory (`corpus`). Echidna will add "markers" for every line to show whether it observed a revert (indicated by `r`) or a successful call (indicated by `*`).

```
    |     ///@notice stop the execution of a long term order
*r  |     function cancelLongTermSwap(uint256 orderId) external lock
execVirtualOrders {
*r  |         (address sellToken, uint256 unsoldAmount, address buyToken, uint256
purchasedAmount) = longTermOrders.cancelLongTermSwap(orderId);
    |
*   |         bool buyToken0 = buyToken == token0;
*   |         twammReserve0 -= uint112(buyToken0 ? purchasedAmount : unsoldAmount);
*   |         twammReserve1 -= uint112(buyToken0 ? unsoldAmount : purchasedAmount);
    |
    |         // transfer to owner of order
*   |         _safeTransfer(buyToken, msg.sender, purchasedAmount);
*   |         _safeTransfer(sellToken, msg.sender, unsoldAmount);
    |
    |         // update order. Used for tracking / informational
*   |         longTermOrders.orderMap[orderId].isComplete = true;
    |
*   |         emit CancelLongTermOrder(msg.sender, orderId, sellToken, unsoldAmount,
buyToken, purchasedAmount);
    |     }
```

Figure E.6: Echidna's markers on the left to show coverage

Monitoring Echidna's coverage is important to ensure that the fuzzing campaign explores all the relevant code. It should be performed regularly before starting a long-term fuzzing campaign.

Integrating Fuzzing into the Development Cycle

Once the fuzzing procedure has been tuned to be fast and efficient, it should be properly integrated in the development cycle to catch bugs. We recommend adopting the following procedure to integrate fuzzing using a continuous integration system:

1. After the initial fuzzing campaign, save the corpora generated from every test.
2. For every internal milestone, new feature, or public release, run the fuzzing campaign again for at least 24 hours, starting with the current corpora for each test.[1]
3. Update the corpora with the new inputs generated.

Note that, over time, the corpora will come to represent thousands of CPU hours of refinement and will be valuable for guiding efficient code coverage during fuzz testing. An

[1] For more on fuzz-driven development, see this CppCon 2017 talk by Kostya Serebryany of Google.

attacker could also use the corpora to quickly identify vulnerable code, so we recommend avoiding this additional risk by keeping fuzzing corpora in an access-controlled storage location rather than a public repository. Some continuous integration systems allow maintainers to keep a cache to accelerate building and testing. The corpora could be included in such a cache, if they are not very large.