Frax Finance Audit Report

(Fraxtal + VestedFXS + Flox)

Frax Security Cartel

0xleastwood, Riley Holterhus, Zach Obront

April 15, 2024

Contents

1 Introduction

1.1 About Frax Finance

Frax Finance is a DeFi industry leader, featuring several subprotocols that support the Frax, FPI, and frxETH stablecoins. In early 2024, Frax also launched Fraxtal - an optimistic rollup built using the OP stack framework. For more information, visit Frax's website: frax.finance.

1.2 About the Auditors

0xleastwood, Riley Holterhus, and Zach Obront are independent smart contract security researchers. All three are Lead Security Researchers at Spearbit, and have a background in competitive audits and live vulnerability disclosures. As a team, they are working together to conduct audits of Frax's codebase, and are operating as the "Frax Security Cartel".

0xleastwood can be reached on Twitter at @0xleastwood, Riley Holterhus can be reached on Twitter at @rileyholterhus and Zach Obront can be reached on Twitter at @zachobront.

1.3 Disclaimer

This report is intended to detail the identified vulnerabilities of the reviewed smart contracts and should not be construed as an endorsement or recommendation of any project, individual or entity. While the authors have made reasonable efforts to detect potential issues, the absence of any undetected vulnerabilities or issues cannot be guaranteed. Additionally, the security of the smart contracts may be affected by future changes or updates. By using the information in this report, you acknowledge that you are doing so at your own risk and that you should exercise your own judgment when implementing any recommendations or making decisions based on the findings. This report has been provided on an "as-is" basis and DOES NOT CONSTITUTE A GUARANTEE OR WARRANTY OF ANY FORM.

2 Audit Overview

2.1 Scope of Work

From February 26th, 2024 through March 8th, 2024, the Frax Security Cartel conducted an audit of three new components of the Frax codebase. During this period, the code was manually analyzed for various security issues and logic flaws. The audit was conducted on Frax's internal dev-fraxchain-contracts GitHub repository, specifically on commit hash bca9bf270acfe950ef3abc82f58d42c6023a5968. In the audit, the following components of the codebase were reviewed:

Fraxtal - the system contracts that are used in Frax's new OP stack rollup. These contracts are designed to support frxETH as the rollup's native gas token. The following directories were in scope for this component:

- contracts/Fraxtal/L1/
- contracts/Fraxtal/L2/
- contracts/Fraxtal/interfaces/
- contracts/Fraxtal/libraries/
- contracts/Fraxtal/universal/

VestedFXS - the contracts used in Frax's voting escrow system. Previous versions of the VestedFXS contracts were written in Vyper, and have been live on mainnet since 2021. The audited version is written in Solidity, and includes new features such as multiple independent locks per user. The following directories were in scope for this component:

- contracts/VestedFXS-and-Flox/VestedFXS/

Flox - the contracts used in the Fraxtal Blockspace Incentives system (referred to as "Flox"). These contracts are used in the process of rewarding and rebating addresses that have spent gas on Fraxtal. The following directories were in scope for this component:

- contracts/VestedFXS-and-Flox/Flox/

2.2 Summary of Findings

Each finding from the audit has been assigned a severity level of "Critical", "High", "Medium", "Low" or "Informational". These severities are somewhat subjective, but aim to capture the impact and likelihood of each potential issue. Gas optimization findings have been included in their own section of the report.

In total, **19 findings** were identified. This includes **1 critical**, **2 high**, **4 medium**, **3 low**, and **8 informational** severity findings, as well as **1 gas optimization** finding. All issues have either been directly addressed by the Frax team, or have been acknowledged as acceptable behavior.

3 Findings

3.1 Critical Severity Findings

3.1.1 Past balance checks can be DOS'd by depositing after target timestamp

Description: The `balanceOfAllLocksAtTime()` function is intended to return a user's total voting power at a given timestamp, including timestamps in the past. In both the new Solidity implementation and the old Vyper implementation, the following code is used during this historical calculation:

```
lastPoint.bias -= lastPoint.slope * int128(uint128(_timestamp - lastPoint.ts));
```

Notice that this code will revert due to underflow if `lastPoint.ts` is newer than `_timestamp`, which may be the case, since `_timestamp` is an arbitrary user input. This implies that the historical calculation will revert for any timestamp older than when the user last interacted with the `veFXS` contract.

This problem may arise naturally for users who have recently deposited into their `veFXS` locks, and it can also be deliberately triggered. Indeed, a third-party attacker can spend a small amount of `FXS` in the `depositFor()` function to update several users' `lastPoint.ts` values. Note that the Vyper implementation's `deposit_for()` function is slightly different, as it requires the user to already have an allowance to the `veFXS` contract (which might be the case with lingering or unlimited approvals).

Currently, the historical lookup functionality is used in Frax's on-chain governance to determine voting powers at the time when a proposal started. An attacker might leverage this problem to DOS other users from voting. This is especially a problem in the FraxGovernorAlpha contract, since an attacker would have less intervention on malicious proposals they submit.

Proof of Concept: The following test can be added to `Unit_Test_VestedFXS.t.sol`. It involves checking the balance of a lock at a time before the lock was last checkpointed. When the test is run, it leads to a revert due to underflow.

```
function testZach_NoPastBalance() public {
    vestedFXSSetup();

    hoax(bob);
    token.approve(address(vestedFXS), 60e18);
    hoax(bob);
    vestedFXS.createLock(bob, 50e18, uint128(block.timestamp + 1 weeks));

    vestedFXS.balanceOfAllLocksAtTime(bob, block.timestamp - 1);
}
```

```
[FAIL. Reason: panic: arithmetic underflow or overflow (0x11)]
```

You can see a full proof of concept of how this can be used to DOS governance here.

Recommendation: Do a binary search lookup for the user's latest point that has a timestamp less than or equal to the relevant timestamp, and use this point for extrapolation (instead of always using the latest point):

```
- Point memory lastPoint = userPointHistory[_addr][lockId][_epoch];
+ uint256 _min = 0;
+ uint256 _max = userPointEpoch[_addr][lockId];
+ for (uint256 i; i < 128; ) {
+     // Will be always enough for 128-bit numbers
+     if (_min >= _max) {
+         break;
+     }
+     uint256 _mid = (_min + _max + 1) / 2;
+     if (userPointHistory[_addr][lockId][_mid].ts <= _timestamp) {
+         _min = _mid;
+     } else {
+         _max = _mid - 1;
+     }
+
+     unchecked {
+         ++i;
+     }
+ }
+ Point memory lastPoint = userPointHistory[_addr][lockId][_min];
  lastPoint.bias -= lastPoint.slope * int128(uint128(_timestamp - lastPoint.ts));
```

This is the solution used by Yearn (see their Vyper implementation here) and Extra Finance (see their Solidity implementation here).

Since it's not possible to modify the Vyper `veFXS` implementation that's already been deployed, an additional recommendation is to adapt any external contracts to use the `balanceOfAt()` function. This function takes a block number as input instead of a timestamp, and does not have the same issues described in this finding.

Frax: Fixed in commit 7cf971c and commit 53c405e.

Frax Security Cartel: Verified.

3.2 High Severity Findings

3.2.1 Past total supply can return incorrect values

Description: The `totalSupply(uint256 _timestamp)` function allows for historical lookups of total voting power, and has the following implementation:

```
function totalSupply(uint256 _timestamp) public view returns (uint256) {
    _timestamp = _timestamp == 0 ? block.timestamp : _timestamp; // Default to current timestamp if
         t is 0
    Point memory lastPoint = pointHistory[epoch];
    return supplyAt(lastPoint, _timestamp);
}
```

Notice that this logic passes `pointHistory[epoch]` (the most recent point) to the `supplyAt()` function, regardless of whether the target `_timestamp` is before or after this point. Later on, the `supplyAt()` function uses the following logic:

```
for (uint256 i; i < 255; ) {
    tI += WEEK_UINT256;
    int128 dSlope = 0;
    if (tI > _t) {
        tI = _t;
    } else {
        dSlope = slopeChanges[tI];
    }
    if (tI >= lastPoint.ts) {
        lastPoint.bias -= lastPoint.slope * int128(uint128(tI - lastPoint.ts));
    } else {
        lastPoint.bias += lastPoint.slope * int128(uint128(lastPoint.ts - tI));
    }
    if (tI == _t) {
        break;
    }
    lastPoint.slope += dSlope;
    lastPoint.ts = tI;

    unchecked {
        ++i;
    }
}
```

In the case where `tI >= lastPoint.ts` is false (which happens when the target timestamp is earlier than the initial `lastPoint.ts`), this logic will extrapolate the `lastPoint` information backward. This can lead to incorrect results, since the `lastPoint` information can contain new deposits that did not exist in the past (but are still included in the historical calculation).

Proof of Concept: The following test can be added to `Unit_Test_VestedFXS.t.sol`. It involves checking the supply at a time when there is no supply, and getting the result of `0`. After the manipulation, we can trick the contract into extrapolating a new supply that is substantially higher than 0.

Note that this attack, if timed properly at the end of an epoch, only requires locking up funds for 1 block.

```
function testZach_totalSupplyBug() public {
    vestedFXSSetup();
    vestedFXS.checkpoint();
    uint targetTs = block.timestamp;

    (uint128 earliest, ) = vestedFXS.getCreateLockTsBounds();
    vm.warp(earliest + 21 days - 1);
    uint supply = vestedFXS.totalSupply(targetTs);
    assertEq(supply, 0);

    hoax(bob);
    token.approve(address(vestedFXS), 50e18);
    hoax(bob);
    vestedFXS.createLock(bob, 50e18, uint128(block.timestamp + 1));

    uint newSupply = vestedFXS.totalSupply(targetTs);
    assertEq(newSupply, 52294436120623716839);
}
```

Recommendation: Instead of always using the most recent point, the `totalSupply()` function should do a binary search to determine the most accurate value in the `pointHistory` to extrapolate from:

```
function totalSupply(uint256 _timestamp) public view returns (uint256) {
    _timestamp = _timestamp == 0 ? block.timestamp : _timestamp; // Default to current timestamp
    if t is 0
+   uint256 _min = 0;
+   uint256 _max = epoch;
+   uint256 _mid;
+   for (uint256 i; i < 128; ) {
+       // Will be always enough for 128-bit numbers
+       if (_min >= _max) {
+           break;
+       }
+       _mid = (_min + _max + 1) / 2;
+       if (pointHistory[_mid].ts <= _timestamp ) {
+           _min = _mid;
+       } else {
+           _max = _mid - 1;
+       }
+
+       unchecked {
+           ++i;
+       }
+   }
+   Point memory lastPoint = pointHistory[_min];
-   Point memory lastPoint = pointHistory[epoch];
    return supplyAt(lastPoint, _timestamp);
}
```

With this change, the code will always extrapolate forward from the `lastPoint`, except when the target timestamp is older than the first entry in `pointHistory` (which is created in the `initialize()` function). Since it would be acceptable for `totalSupply()` to revert on timestamps older than when `VestedFXS` was initialized, the `supplyAt()` function could also be simplified to only support forward extrapolation:

```
function supplyAt(Point memory _point, uint256 _t) public view returns (uint256) {
    Point memory lastPoint = _point;
    uint256 tI = (lastPoint.ts / WEEK_UINT256) * WEEK_UINT256;
+   require(lastPoint.ts <= _t);
    // ...
-   if (tI >= lastPoint.ts) {
        lastPoint.bias -= lastPoint.slope * int128(uint128(tI - lastPoint.ts));
-    } else {
-        lastPoint.bias += lastPoint.slope * int128(uint128(lastPoint.ts - tI));
-    }
    // ...
}
```

Frax: Fixed in PR 62.

Frax Security Cartel: Verified.

3.2.2 Smart contract wallets bridging frxETH will lead to loss of all funds

Description: When users want to bridge `frxETH` directly to Fraxtal (rather than bridging L1 ETH and having it converted into frxETH), they interact directly with `FraxchainPortal.sol`. Specifically, the `bridgeFrxETH()` function, which allows them to transfer `frxETH` into the contract, and then calls the deposit function, overriding the `msg.value` with the amount of `frxETH` sent.

```
function bridgeFrxETH(uint256 _value) external {
    depositTransactionWithFrxETH(msg.sender, _value, RECEIVE_DEFAULT_GAS_LIMIT, false, bytes(""));
}

function depositTransactionWithFrxETH(
    address _to,
    uint256 _value,
    uint64 _gasLimit,
    bool _isCreation,
    bytes memory _data
) public {
    require(
        IERC20(FRXETH).transferFrom(msg.sender, address(this), _value),
        "FraxchainPortal: frxETH transfer failed"
    );
    depositTransaction_internal(_to, _value, _value, _gasLimit, _isCreation, _data);
}
```

As we can see, this function sets `msg.sender` as the `_to` value.

However, when transactions are bridged from L1 to L2, we transform any contract addresses into an “alias”. This is intended to ensure that L2 contracts with different owners from their L1 counterparts don’t end up with control over bridged funds.

We can see this logic here:

```
address from = msg.sender;
if (msg.sender != tx.origin) {
    from = AddressAliasHelper.applyL1ToL2Alias(msg.sender);
}
```

However, the `_to` address is never aliased, and will have the funds sent to it directly.

As a result, any smart contract wallet that uses the `bridgeFrxETH()` function will send funds to their L1 address on Fraxtal, but they will not have control over that address. Instead, they have control over the aliased version (because they can make calls through the bridge where the `from` is set to their aliased address). The result is that their funds will be permanently lost.

Proof of Concept: The following test demonstrates that, when an address with code on L1 (in this case used a Gnosis Safe) calls the `bridgeFrxETH()` function, the funds are sent `from` the aliased address, but `to` the original address.

```
//SPDX-License-Identifier: Unlicense
pragma solidity ^0.8.0;

import "./BaseTestL2.t.sol";
import { AddressAliasHelper } from "@eth-optimism/contracts-bedrock/src/vendor/AddressAliasHelper.
    sol";
import { ERC20PermitPermissionedOptiMintable as IERC20 } from "src/contracts/Fraxtal/universal/
    ERC20PermitPermissionedOptiMintable.sol";

interface IFraxchainPortal {
    function bridgeFrxETH(uint256 _value) external;
}

contract DepositWithSmartWalletTest is BaseTestL2 {
    IFraxchainPortal portal = IFraxchainPortal(0x36cb65c1967A0Fb0EEE11569C51C2f2aA1Ca6f6D);
    IERC20 FRXETH = IERC20(0x5E8422345238F34275888049021821E8E08CAa1f);
    address FRXETH_MINTER = 0xbAFA44EFE7901E04E39Dad13167D089C559c1138;
    event TransactionDeposited(address indexed from, address indexed to, uint256 indexed version,
        bytes opaqueData);

    address safe = 0xAec5a0EB25232d25c9A2D86146d6ABCDEB490B99;

    function testBridgeWithSmartWallet() public {
        vm.createSelectFork("https://eth.llamarpc.com/");

        vm.prank(FRXETH_MINTER);
        FRXETH.minter_mint(safe, 1e18);

        vm.startPrank(safe);
        FRXETH.approve(address(portal), 1e18);

        vm.expectEmit(true, true, false, false);
        emit TransactionDeposited(AddressAliasHelper.applyL1ToL2Alias(safe), safe, 0, bytes(""));
        portal.bridgeFrxETH(1e18);
    }
}
```

Recommendation: The cleanest option is to emulate what Optimism does via their bridge, which is to only allow EOAs:

```
/// @notice Only allow EOAs to call the functions. Note that this is not safe against contracts
///         calling code within their constructors, but also doesn't really matter since we're
///         just trying to prevent users accidentally depositing with smart contract wallets.
modifier onlyEOA() {
    require(!Address.isContract(msg.sender), "StandardBridge: function can only be called from an
        EOA");
    _;
}
```

Frax: Acknowledged.

Frax Security Cartel: Acknowledged. Following a discussion with the team, it was determined that users directly using the portal are expected to have the knowledge to avoid this error. It is preferred to not add an `onlyEOA()` modifier to `bridgeFrxETH()`, because this would imply the need for a similar change to the `receive()` function, and this would be a deviation from the original Optimism interface.

In the future, new contracts built on top of the portal could add the EOA check before calling `bridgeFrxETH()`. This idea does not require immediate changes to current contracts.

3.3 Medium Severity Findings

3.3.1 Users may not be able to deposit assets after withdrawing

Description: The `createLock()` function enables users to create their own locks, provided they have fewer than 8 locks in total. On the other hand, whitelisted `floxContributors` can create locks for users with fewer than 16 total locks. This is facilitated by the following code, where `MAX_USER_LOCKS == 8` and `MAX_CONTRIBUTOR_LOCKS == 16`:

```
if (
    (!floxContributors[msg.sender] && numLocks[_addr] >= MAX_USER_LOCKS) ||
    numLocks[_addr] >= MAX_CONTRIBUTOR_LOCKS
) revert MaximumLocksReached();
```

With this logic, it's possible that a user can be in a state where they are not able to make any deposits. This can happen before the user creates any locks of their own, because a `floxContributor` can fill up a user's first 8 locks.

Moreover, even if a user does have their own locks, withdrawing from any of them will not necessarily allow them to deposit again. This may be unexpected for users, as if they have more than 8 locks (no matter who created them), no single withdrawal will allow them to create another lock.

Recommendation: It is recommended to maintain separate counters or mappings for user and contributor locks. This separation will help ensure that users are never in scenarios where they are unexpectedly unable to deposit.

Frax: Differentiated the user and contributor counts in PR 59.

Frax Security Cartel: Verified.

3.3.2 Mutable `numLocks` value leads to incorrect past balances

Description: When accessing the past balance of a user, we retrieve the number of locks they have and iterate over each lock to get the past balance, summing the results.

```
function balanceOfAllLocksAtTime(address _addr, uint256 _timestamp) public view returns (uint256
    _balance) {
    // Get the total number of locks
    uint128 _numLocks = numLocks[_addr];

    // Loop through all of the locks
    for (uint128 i = 0; i < _numLocks; ) {
        _balance += balanceOfOneLockAtTime(_addr, i, _timestamp);

        unchecked {
            ++i;
        }
    }
}
```

However, the `numLocks` value does not represent the number of locks at that point in the past. It represents the current number of locks, which decreases when a lock is withdrawn.

```
// Update numLocks
numLocks[_staker] -= 1;
```

As a result, any withdrawn locks will not be included in the past balance calculation, leading to incorrect results.

Proof of Concept: The following test can be added to `Unit_test_VestedFXS.t.sol` to demonstrate that we get different values for a specific past balance depending on the current locks.

```
function testZach_PastBalance() public {
    vestedFXSSetup();

    hoax(bob);
    token.approve(address(vestedFXS), 60e18);
    hoax(bob);
    vestedFXS.createLock(bob, 50e18, uint128(block.timestamp + 1 weeks));

    uint targetTs = block.timestamp;
    assertApproxEqAbs(vestedFXS.balanceOfAllLocksAtTime(bob, targetTs), 50e18, 1e18);

    skip(1 weeks);
    hoax(bob);
    vestedFXS.withdraw(0);
```

```
        assertEq(vestedFXS.balanceOfAllLocksAtTime(bob, targetTs), 0);
    }
```

Recommendation: After a discussion with the Frax team, it was decided that this is acceptable behavior, since a withdrawal from a lock only removes the historical voting power associated with that specific lock. So, the recommendation is to keep this behavior in mind, especially for codebases that use historical voting power calculations.

Frax: As described above, acknowledged.

Frax Security Cartel: Acknowledged.

3.3.3 Contracts with failed frxETH deposits may be unable to access funds

Description: When `frxETH` is deposited from L1 to L2, the Portal is called directly with the `depositTransactionWithFrxETH()` function.

```
function depositTransactionWithFrxETH(
    address _to,
    uint256 _value,
    uint64 _gasLimit,
    bool _isCreation,
    bytes memory _data
) public {
    require(
        IERC20(FRXETH).transferFrom(msg.sender, address(this), _value),
        "FraxchainPortal: frxETH transfer failed"
    );
    depositTransaction_internal(_to, _value, _value, _gasLimit, _isCreation, _data);
}
```

Note that all deposits performed directly to the Portal are non-replayable. Replayability across the bridge comes from the Cross Domain Messenger, and therefore failed deposit transactions will simply fail.

In order to handle this issue, in the event of a failed transaction, the system will mint the correct amount of the native token to the `from` address on L2.

However, in the case of contracts calling this function, the `from` address will be an alias of their address:

```
// Transform the from-address to its alias if the caller is a contract.
address from = msg.sender;
if (msg.sender != tx.origin) {
    from = AddressAliasHelper.applyL1ToL2Alias(msg.sender);
}
```

In this case, the only way to access the L2 funds would be to make another call through the Portal, with the calldata necessary to return the funds (or direct them where they should be on L2). However, many contracts do not have the functionality built in to perform this specific call. As a result, their funds will be stuck in their aliased L2 address.

Recommendation: There are a few potential solutions that could help with this issue. The easiest would be to leave the contracts as-is, and this risk can simply be documented to ensure that users are careful. A more involved solution would be to only allow EOAs to use this function, so that contracts are forced to go through the bridge with native ETH. A final change could be to move some of the related logic to the Cross Domain Messenger, so it's replayable on the other side.

Frax: This is a low level function, so it will only be used by users/contracts that know what they are doing. So I would go for just adding some documentation.

Frax Security Cartel: Acknowledged.

3.3.4 Delegation for Cross Domain Messenger can be claimed to steal FXTL points

Description: The `DelegationRegistry` is used to allow contracts to assign a delegate. This address will receive FXTL points on their behalf, to ensure that contracts without the ability to perform arbitrary calls are not left unable to access their points.

To this end, the contract can call `setDelegationForSelf()` on the Registry contract, which will assign a new `delegatee` for their address, who is given their points.

However, any user can make the `CrossDomainMessenger` contract perform arbitrary external calls. This can be done by bridging a message from L1 to L2 with the `DelegationRegistry` as the `target`, and the `setDelegationForSelf` function call as the calldata.

Since the `CrossDomainMessenger` is a very popular contract that will have a large amount of gas used on it, it is likely to be allocated a large proportion of the FXTL points. Any user that sets themselves as the delegatee before the calculation is done will end up receiving those points, stealing future rewards from other users.

Note on Impact: While this issue is clearly present in the Cross Domain Messenger, the same issue holds for all contracts that allow any user to perform arbitrary external calls. It is important for the Fraxtal team to make developers aware of this issue so they can limit risk.

Recommendation: To solve the problem for the Cross Domain Messenger, it is recommended to blacklist all Predeploy contracts from receiving any portion of the FXTL points. This can be handled in the offchain script.

To solve the larger problem, it is necessary to allow contracts to lock their delegation. That way, susceptible contracts can delegate and lock from their constructors to ensure that they are no longer vulnerable.

[Note that if the claiming logic will only allow claiming to another address, then it is extra important that these vulnerable contracts delegate to another address instead of claiming themselves, as these claims would be vulnerable to the same attack.]

Frax: Fixed in PR 56.

Frax Security Cartel: Verified.

3.4 Low Severity Findings

3.4.1 Tokens minted on L2 could lead to bridge insolvency

Description: The `ERC20PermitPermissionedOptiMintable` token is a modified version of the Optimism token with permissions added for non-bridge actors to mint or burn tokens. This is intended to mirror Frax's permissions over their tokens on L1, and this implementation is used for all Frax predeploy tokens (`FPI`, `FPIS`, `FRAX`, `frxBTC`, `FXS`, `sFRAX`, `sfrxETH`, and `wfrxETH`).

However, minting more tokens on L2 than exist in the bridge on L1 leads to bridge insolvency.

While it may appear on L2 as if more tokens are appearing, the reality is that if all users chose to withdraw their L2 tokens, there wouldn't be enough L1 tokens in the bridge to fulfill the requirements. The first users would receive the L1 tokens back, while the final users would not be able to withdraw.

Recommendation: Rather than minting directly on L2, it would be preferable to mint on L1 and bridge the tokens to L2. This will ensure that (a) the bridge is adequately funded and (b) each token respects its L1 supply cap.

Convenience functions could be added on L1 to facilitate this simple mint, bridge & distribute flow in one transaction.

Frax: Acknowledged. The team will monitor token balances to ensure solvency is maintained.

Frax Security Cartel: Acknowledged.

3.4.2 Minters cannot burn tokens unless user provides them with allowance

Description: The `ERC20PermitPermissionedOptiMintable` contract is a modified token that gives a specified list of minters the power to mint or burn tokens.

```
function minter_burn_from(address b_address, uint256 b_amount) public onlyMinters {
    super.burnFrom(b_address, b_amount);
    emit TokenMinterBurned(b_address, msg.sender, b_amount);
}

function minter_mint(address m_address, uint256 m_amount) public onlyMinters {
    super._mint(m_address, m_amount);
    emit TokenMinterMinted(msg.sender, m_address, m_amount);
}
```

The Frax team has confirmed that these minters should be able to arbitrarily mint or burn tokens.

This works correctly for minting because the internal `_mint()` function is called directly. This bypasses all checks and mints the tokens to the specified address.

For the burn function, however, we do not call the internal `_burn()` function directly. Instead, we call the `burnFrom()` function, which requires the user to have approved the minter to burn their tokens.

```
function burnFrom(address account, uint256 amount) public virtual {
    _spendAllowance(account, _msgSender(), amount);
    _burn(account, amount);
}
```

As a result, any attempt for the minters to burn user tokens (unless the user has given approval) will revert.

Proof of Concept: The following test demonstrates that the minter is free to mint tokens, but cannot burn tokens without the user's approval.

```
function testBurnFrom() public {
    vm.createSelectFork("https://rpc.frax.com");
    address minter = makeAddr("minter");
    address user = makeAddr("user");

    ERC20PermitPermissionedOptiMintable FRAX = ERC20PermitPermissionedOptiMintable(FRAXTAL_L2_FRAX)
        ;
    vm.prank(FRAXTAL_L2_FRAXTAL_SAFE);
    FRAX.addMinter(minter);
    assertEq({ a: FRAX.minters(minter), b: true, err: "Minter not added to erc20" });

    vm.startPrank(minter);
    FRAX.minter_mint(user, 1e18);

    vm.expectRevert(abi.encodeWithSelector(IERC20Errors.ERC20InsufficientAllowance.selector, minter
        , 0, 1e18));
    FRAX.minter_burn_from(user, 1e18);
}
```

Recommendation:

```
function minter_burn_from(address b_address, uint256 b_amount) public onlyMinters {
-   super.burnFrom(b_address, b_amount);
+   super._burn(b_address, b_amount);
    emit TokenMinterBurned(b_address, msg.sender, b_amount);
}
```

The following modified test (using a deployed token rather than the forked chain) demonstrates that the minter can now burn tokens without the user's approval.

```
function testBurnFrom() public {
    address minter = makeAddr("minter");
    address user = makeAddr("user");

    ERC20PermitPermissionedOptiMintable token = new ERC20PermitPermissionedOptiMintable(
        FRAXTAL_L2_FRAXTAL_SAFE,
        address(1),
        address(1),
        address(1),
        "Test Token",
        "TEST"
    );
    vm.prank(FRAXTAL_L2_FRAXTAL_SAFE);
    token.addMinter(minter);
    assertEq({ a: token.minters(minter), b: true, err: "Minter not added to erc20" });

    vm.startPrank(minter);
    token.minter_mint(user, 1e18);

    // vm.expectRevert(abi.encodeWithSelector(IERC20Errors.ERC20InsufficientAllowance.selector,
        minter, 0, 1e18));
    token.minter_burn_from(user, 1e18);
}
```

Frax: Unfortunately the already-deployed FRAX, FXS, etc main tokens on Fraxtal have this "bug", but it is not really used much, or severe enough to warrant a redeploy. Will fix for new tokens going forward.

Frax Security Cartel: Acknowledged.

3.4.3 BalanceChecker returns incorrect values for ETH balance

Description: The `BalanceChecker` contract provides helper functions used to check the balance of various tokens on an address (or various addresses for a given token).

It has functionality added to access ETH balances. Specifically, when `token == address(0)`, the usual flow of checking `balanceOf()` is overridden.

However, rather than pulling the ETH balance of the address, we instead simply return 0.

Recommendation:

```diff
if (token == address(0)) {
    for (uint256 i = 0; i < addresses.length; ++i) {
-       result[i] = 0;
+       result[i] = addresses[i].balance;
    }
    return result;
}
```

```diff
for (uint256 i = 0; i < tokens.length; ++i) {
-   if (tokens[i] == address(0)) result[i] = 0;
+   if (tokens[i] == address(0)) result[i] = addr.balance;
    else result[i] = IERC20(tokens[i]).balanceOf(addr);
}
```

Frax: Fixed in PR 58.

Frax Security Cartel: Verified.

3.5 Informational Findings

3.5.1 Naming improvements for `idsToIndices` mapping

Description: The `idsToIndices` importantly keeps track of a user's lock global and local index. Ultimately when a lock is created, the `indicesToIds` array is populated with a new unique index, representing all locks even if expired but yet to be withdrawn. When a lock is removed from the array, it needs to be popped off the `indicesToIds` array and deleted from the `idsToIndices`.

It is a little unclear on the distinguished role of the values in the `LockIdIdxInfo` struct. The naming for `id` and `index` is not apparent until it is seen how they are used in the contract implementation. The naming for these values can be improved such that it is clear from looking at the contract interface what role they play.

Recommendation: Consider renaming struct values in `LockIdIdxInfo` to be more representative of how global and local indices are managed by the contract.

Frax: Acknowledged.

Frax Security Cartel: Acknowledged.

3.5.2 `FraxchainPortal` blacklisted target considerations

Description: In the `FraxchainPortal`, the `finalizeWithdrawalTransaction()` function reverts if the `_tx.target` is equal to `frxETH`. This is a crucial check. Without it, an attacker could call the token address with arbitrary calldata in a withdrawal, which would allow a theft of the `frxETH`.

In addition to this check, it should be considered whether or not the portal's `frxETH` can be burned/transferred by targeting some other contract in the Frax ecosystem. After checking, this is *not* possible, because the `frxETH` contract itself does not allow token movement without a prior approval. Even the `minter_burn_from()` function requires that the `minter` has been approved to burn the user's tokens. Since `frxETH` is blacklisted as a target (and since a valid `permit()` signature would be impossible), the portal will never approve any address, which resolves this concern.

A final consideration is a scenario where the portal receives special powers simply due to its balance of `frxETH`. For example, if `frxETH` had a voting power associated with it, it'd be important to either blacklist the voting contract as a target, or, the voting contract should exclude the portal from its calculations. There does not appear to be any live system where this consideration matters, but it would be useful to be aware of this in the future.

Recommendation: As the Frax ecosystem grows, keep in mind that the `FraxchainPortal` has a large `frxETH` balance, and allows arbitrary calls to essentially all addresses. If any future contracts utilize a user's `frxETH` balance for special privileges, consider excluding the `FraxchainPortal` from any calculations in this hypothetical system.

Frax: Acknowledged.

Frax Security Cartel: Acknowledged.

3.5.3 Emergency unlock accounting can be improved

Description: When `emergencyUnlockActive` is true, users will likely choose to withdraw from their locks early. However, this is not explicitly required, and there may still be expired withdrawals that happen during emergency unlocks. In this case, the following section of code will not be entered:

```
if (_newLocked.amount == 0 && !emergencyUnlockActive) {
    lastPoint.bias -= _oldLocked.amount;
}
```

However, in this specific circumstance, the internal accounting would be more correct if it were entered. Indeed, even if `emergencyUnlockActive` is true, an expired withdrawal has the same result on the accounting, and the bias should be decremented.

Moreover, it may be preferable to introduce hardcoded logic in the public view functions when `emergencyUnlockActive` is true. For example, the `lockedEnd()` function could always return `block.timestamp` when `emergencyUnlockActive` is true, since this extraordinary scenario will have permanently changed the logic of the contract.

Recommendation: Change the code to skip the bias decrement only when the withdrawal is early. This matches the logic in other parts of the code, as this bias decrement is accounted for elsewhere.

```
if (_newLocked.amount < _oldLocked.amount) {
    lastPoint.fxsAmt -= uint256(uint128(_oldLocked.amount - _newLocked.amount));

-   if (_newLocked.amount == 0 && !emergencyUnlockActive) {
+   if (_newLocked.amount == 0 && _oldLocked.end <= block.timestamp) {
        lastPoint.bias -= _oldLocked.amount;
    }
}
```

Also, consider introducing special cases in the public view functions when `emergencyUnlockActive` is true, so that the internal accounting is not heavily relied on in times of emergency.

Frax: Fixed the bias decrement in commit d9f54bf. Added the `emergencyUnlockActive` special cases in PR 60.

Frax Security Cartel: Verified.

3.5.4 L2 to L1 messages require twice the ETH

Description: There are two types of L2 withdrawals that can be made through the `FraxchainPortal`. If a withdrawal contains a `_tx.data` equal to `bytes1(0xfe)`, it's treated as a simple `frxETH` transfer to the withdrawal recipient. Otherwise, the withdrawal is treated as a typical L2 to L1 message, possibly with an associated `msg.value`. However, since the `mintFrxETH()` function in the `FraxchainPortal` allows anyone to convert the contract's ETH balance into `frxETH`, it's likely that the portal will not have the required ETH for `msg.value` purposes.

To mitigate this, the `finalizeWithdrawalTransaction()` function has been marked as `payable`, and any ETH provided to the function will be sent back to the caller as `frxETH`. This allows anyone to execute messages that are missing the required L1 ETH, which provides a path to solve the problem.

In cases where a protocol lacks the necessary L1 ETH to complete a cross-chain message, using an ETH flashloan might be a solution. This loan could be repaid after swapping the `frxETH` received from the portal, although users should be aware that this method would involve fees and slippage.

Recommendation: Consider documenting this behavior for external developers that are planning to do cross-chain messaging. Also, if the current behavior leads to any issues in the future, consider implementing a long-term solution. For example, it might be possible to incentivize users to swap ETH for `frxETH` using the `finalizeWithdrawalTransaction()` function. Another possible solution would be to initiate an official `frxETH` to `ETH` withdrawal in the `proveWithdrawalTransaction()` function, however note that this could increase withdrawal times.

Frax: Acknowledged.

Frax Security Cartel: Acknowledged.

3.5.5 No helper function to safely withdraw L2 ETH as frxETH

Description: When withdrawing L2 ETH to L1, we have the option to withdraw `frxETH` on L1 instead of native ETH. This is done by passing a `data` value of `0xfe` into the withdrawal.

```
// when data is 0xfe, we transfer frxETH instead of ETH
bool isFrxETHTransfer = _tx.data.length == 1 && bytes1(_tx.data) == bytes1(0xfe);
```

Since many users will be bridging their `frxETH` on to L2, it is expected that many will want to withdraw in this way as well. It also simplifies the withdrawal UX, since there is no need to provide a `msg.value` to make the withdrawal succeed.

However, there is presently no easy and safe way to perform this withdrawal. While ETH can be withdrawn using the `L2StandardBridge`, because it will have the data needed to perform the bridge withdrawal, it will be withdrawn as ETH.

The only way to perform the `frxETH` withdrawal is to manually call the `L2ToL1MessagePasser` with the calldata. This calldata can be unintuitive. For example, `target` is the address to send the funds to, not the address to call (as with other withdrawals), and if they accidentally input `frxETH` as the `target`, funds will be permanently lost.

Recommendation: It would be helpful to adjust the `L2StandardBridge` to provide a helper function that calls `L2ToL1MessagePasser` with the correct data for a `frxETH` withdrawal.

Frax: Acknowledged.

Frax Security Cartel: Acknowledged. It was also discussed that a new set of contracts could be created for this specific purpose. This would not affect the currently deployed contracts, and this may be worth revisiting in the future.

3.5.6 `getIncreaseUnlockTimeTsBounds()` can provide reverting values

Description: VestedFXS provides a `getIncreaseUnlockTimeTsBounds()` to get the earliest and latest acceptable timestamp values for which you can extend a lock.

```
function getIncreaseUnlockTimeTsBounds(
    address _user,
    uint256 _id
) external view returns (uint128 _earliestLockEnd, uint128 _latestLockEnd) {
    // Calculate the earliest end (current end + leftover time to get into next epoch week)
    _earliestLockEnd = WEEK_UINT128 + ((locked[_user][_id].end / WEEK_UINT128) * WEEK_UINT128);

    // Calculate the latest end (same as getCreateLockTsBounds result)
    _latestLockEnd = (WEEK_UINT128 * (uint128(block.timestamp) + uint128(MAXTIME_UINT256))) /
        WEEK_UINT128;

    // Corner case near expiry
    if (_earliestLockEnd >= _latestLockEnd) _earliestLockEnd = _latestLockEnd;
}
```

The corner case handled at the end will occur when a lock is in its first week. In this case, the function will revert the week exactly 4 years in the future, which is the same as the current expiry of the lock.

If a lock is extended with this value, the call will revert with the following:

```
if (unlockTime <= _locked.end) revert MustBeInAFutureEpochWeek();
```

Interacting protocols could rely on the fact that using the bounds provided by `getIncreaseUnlockTimeTsBounds()` will succeed, which could cause problems.

Proof of Concept: The following test can be dropped in to `Unit_test_VestedFXS.t.sol` to demonstrate the issue:

```
function testZach_FirstWeekExtendFails() public {
    vestedFXSSetup();
    vm.warp(block.timestamp + 100);

    hoax(bob);
    token.approve(address(vestedFXS), 50e18);
    (, uint128 latest) = vestedFXS.getCreateLockTsBounds();
    hoax(bob);
    vestedFXS.createLock(bob, 50e18, latest);
    uint256 lockId = vestedFXS.indicesToIds(bob, 0);
    (, uint end) = vestedFXS.locked(bob, lockId);

    vm.warp(block.timestamp + 1 days);
    (uint128 earliest, uint128 late) = vestedFXS.getIncreaseUnlockTimeTsBounds(bob, 0);

    vm.expectRevert(VestedFXS.MustBeInAFutureEpochWeek.selector);
```

```
        hoax(bob);
        vestedFXS.increaseUnlockTime(earliest, 0);

        vm.expectRevert(VestedFXS.MustBeInAFutureEpochWeek.selector);
        hoax(bob);
        vestedFXS.increaseUnlockTime(late, 0);
    }
```

Recommendation: Document for interacting protocols that this function cannot be relied upon to always give bounds that will succeed.

Frax: Added documentation of this behavior in commit 2f8fb05.

Frax Security Cartel: Verified.

3.5.7 `futureAdmin` isn't cleared on VestedFXS ownership transfer

Description: The `VestedFXS` contract uses a two step ownership transfer, where a `futureAdmin` is set by the current admin, and then ownership is claimed by the `futureAdmin`.

```
function acceptTransferOwnership() external {
    if (msg.sender != futureAdmin) revert FutureAdminOnly();
    address _admin = futureAdmin;
    if (_admin == address(0)) revert AdminNotSet(); // This is now unreachable, but I don't mind
        leaving it for the extremely remote chance that someone figures out a way to execute calls
        from 0x0
    admin = _admin;
    emit ApplyOwnership(_admin);
}
```

In this implementation, the `futureAdmin` slot is not cleared after the ownership transfer is completed.

Recommendation: Similar to how transfers are handled in `OwnedV2.sol`, `futureAdmin` should be cleared after the ownership transfer is completed:

```
function acceptTransferOwnership() external {
    if (msg.sender != futureAdmin) revert FutureAdminOnly();
    address _admin = futureAdmin;
    if (_admin == address(0)) revert AdminNotSet();
    admin = _admin;
+   futureAdmin = address(0);
    emit ApplyOwnership(_admin);
}
```

Frax: Fixed in commit 991d094.

Frax Security Cartel: Verified.

3.5.8 `getCreateLockTsBounds()` returns inaccurate latest lock

Description: The `getCreateLockTsBounds()` helper function is intended to return the earlier lock end and latest lock end that a user could currently perform for a fresh lock.

```
function getCreateLockTsBounds() external view returns (uint128 _earliestLockEnd, uint128
    _latestLockEnd) {
    _earliestLockEnd = WEEK_UINT128 + ((uint128(block.timestamp) / WEEK_UINT128) * WEEK_UINT128);
    _latestLockEnd = (WEEK_UINT128 * (uint128(block.timestamp) + uint128(MAXTIME_UINT256))) /
        WEEK_UINT128;
}
```

The earliest lock is calculated correctly, by rounding the current timestamp down the last week boundary and adding a week. Rounding down is performed by dividing and then multiplying by the week length.

However, the latest lock end is calculated differently. We add 4 years to the current timestamp, and then (rather than dividing and then multiplying to round down), we multiply and then divide, which doesn't do anything.

The result is that the latest lock end value is exactly 4 years after the current timestamp, rather than the actual value of the latest possible lock end (which should round down to the last week boundary).

Proof of Concept: The following test can be added to `Unit_Test_VestedFXS.t.sol`. We get the `latest` value from this function and then see that the actual `end` returned by locking is less than this value.

Further, we validate the suggested fix, showing that `should` returns the correct value.

```
function testZach_CreateLockTsBounds() public {
    vestedFXSSetup();
    vm.warp(block.timestamp + 100);

    hoax(bob);
    token.approve(address(vestedFXS), 50e18);
    (, uint128 latest) = vestedFXS.getCreateLockTsBounds();
    hoax(bob);
    vestedFXS.createLock(bob, 50e18, latest);

    uint256 lockId = vestedFXS.indicesToIds(bob, 0);
    (, uint end) = vestedFXS.locked(bob, lockId);
    assertLt(end, latest);

    uint256 MAXTIME_UINT256 = 4 * 365 * 86_400;
    uint128 WEEK_UINT128 = 7 * 86_400;
    uint should = ((uint128(block.timestamp) + uint128(MAXTIME_UINT256)) / WEEK_UINT128) *
        WEEK_UINT128;
    assertEq(should, end);
}
```

Recommendation:

```
function getCreateLockTsBounds() external view returns (uint128 _earliestLockEnd, uint128
    _latestLockEnd) {
    _earliestLockEnd = WEEK_UINT128 + ((uint128(block.timestamp) / WEEK_UINT128) * WEEK_UINT128);
-   _latestLockEnd = (WEEK_UINT128 * (uint128(block.timestamp) + uint128(MAXTIME_UINT256))) /
    WEEK_UINT128;
+   _latestLockEnd = ((uint128(block.timestamp) + uint128(MAXTIME_UINT256)) / WEEK_UINT128) *
    WEEK_UINT128;
}
```

Frax: Fixed in commit 61993bf.

Frax Security Cartel: Verified.

3.6 Gas Optimizations

3.6.1 Unnecessary check in `balanceOfOneLockAtBlock()`

Description: VestedFXS `balanceOfOneLockAtBlock()` function gets the historical balance of a lock using the block number.

After finding the `bias`, it performs the following final adjustment before returning the balance:

```
if ((upoint.bias >= 0) || (upoint.fxsAmt >= 0)) {
    if (upoint.bias < int256(upoint.fxsAmt)) _balance = upoint.fxsAmt;
    else _balance = uint256(uint128(upoint.bias));
} else {
    return 0;
}
```

Since `fxsAmt` is a `uint256`, the second condition in the if statement will always pass, which means that path will always be taken. As a result, this check can be stripped out and the logic remains the same.

Recommendation:

```
- if ((upoint.bias >= 0) || (upoint.fxsAmt >= 0)) {
     if (upoint.bias < int256(upoint.fxsAmt)) _balance = upoint.fxsAmt;
     else _balance = uint256(uint128(upoint.bias));
- } else {
-    return 0;
- }
```

Frax: Fixed in commit d2612d1 and commit 52c54d5.

Frax Security Cartel: Verified.